NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

**SIMULATION OF THE
AUTONOMOUS COMBAT SYSTEMS ROBOT
OPTICAL DETECTION SYSTEM**

by
William B. McNeal

December, 1997

Thesis Advisor: Gordon Schacher
Co-Advisor: Don Brutzman

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Master's Thesis

4. TITLE AND SUBTITLE
SIMULATION OF THE AUTONOMOUS COMBAT SYSTEMS ROBOT OPTICAL DETECTION SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
McNeal, William Ben

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT

NPS Combat Systems students learn systems engineering through a series of courses in design, development, implementation, and testing and evaluation. In the last of this series of courses, students design an autonomous robot capable of searching, acquiring, and tracking another autonomous robot having similar capabilities. The project culminates in the Robot Wars Competition, where groups of students have their robots battle each other.

This thesis is the second in a series designed to realistically simulate the robot wars battles. The end-to-end functionality of the optical detection system is modeled, and the necessary physics are implemented for effective simulation and depiction. The model uses a transfer function approach and includes all physical processes, from initial optical beacon emission to final digital control signal. Exercising the model over time using realistic robot inputs yields a simulation that closely replicates real behavior. A Virtual Reality Modeling Language (VRML) program uses data files of each Simbot's movement to generate a 3-dimensional animated scene of the detection sequence. This implemented optical model effectively simulates the SE 3015 robot optical detection system and can reproduce an actual detection and tracking sequence between two robots.

14. SUBJECT TERMS	15. NUMBER OF PAGES
simulation, robots, modeling	151
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev.2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

SIMULATION OF AN OPTICAL DETECTION SYSTEM FOR THE AUTONOMOUS COMBAT SYSTEMS ROBOT

William B. McNeal
Lieutentant, United States Navy
B.S., Southern University, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

ABSTRACT

NPS Combat Systems students learn systems engineering through a series of courses in design, development, implementation, and testing and evaluation. In the last of this series of courses, students design an autonomous robot capable of searching, acquiring, and tracking another autonomous robot having similar capabilities. The project culminates in the Robot Wars Competition, where groups of students have their robots battle each other.

This thesis is the second in a series designed to realistically simulate the robot wars battles. The end-to-end functionality of the optical detection system is modeled, and the necessary physics are implemented for effective simulation and depiction. The model uses a transfer function approach and includes all physical processes, from initial optical beacon emission to final digital control signal. Exercising the model over time using realistic robot inputs yields a simulation that closely replicates real behavior. A Virtual Reality Modeling Language (VRML) program uses data files of each Simbot's movement to generate a 3-dimensional animated scene of the detection sequence. This implemented optical model effectively simulates the SE 3015 robot optical detection system and can reproduce an actual detection and tracking sequence between two robots.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to offer my deepest gratitude to my wife Amanda, daughter Arin and mother Bonnie McNeal for their continued prayers, support, and encouragement. I would also like to thank my advisors, Gordon Schacher and Don Brutzman. Without their assistance, encouragement, and knowledge this project could have never been successfully undertaken and accomplished.

I. INTRODUCTION

A. OVERVIEW

The major focus of the Systems Engineering course, SE 3015, is the system engineering, design, testing, and evaluation of an autonomous robot capable of searching, acquiring, tracking, and attacking another autonomous robot having similar capabilities. The course itself is the capstone in a series of four systems engineering courses designed primarily for students pursuing a Master's Degree in the Combat Systems Science and Technology curriculum. While the course's major emphasis lies in the area of microprocessor architectures, digital communications, and digital and analog interfacing, it also addresses electromechanical systems used for closed-feedback control of positioning, tracking, and detection systems in autonomous robots. It is the end-to-end physical simulation of the closed-feedback positioning, tracking, and detection systems that serves as the focal point for this thesis research.

The positioning, detection, and tracking hardware issued for use with the autonomous robot in the SE 3015 course includes the robot base platform, a small microprocessor called a Tiny Giant, an LED beacon, and two photodiodes. The robot base platform contains two dual drive servo-motors which give the robot movement capability. The base platform includes several I/O ports which allow for analog to digital (A/D) conversion of external data which may be used for robot control. The LED beacon acts as a transmitter and is the device which each robot is designed to detect and track. Each beacon can be set to flash at a unique frequency within the 5-25 kHz range. In turn, students are each assigned one of the unique frequencies for his/her robot to track. Each

student is responsible for designing and building a detector which is composed of an electrical circuit tuned to the specific frequency for which their beacon flashes. The detector consists of a set of photodiodes, which, upon being exposed to a beacon at the tuned flash frequency, supply a triggering voltage that is routed to ports on the Tiny Giant, which perform A/D conversion. A personal computer connected to the Tiny Giant via an RS-232 cable may be used to view the newly converted digital reading through execution of a small program written in the Dynamic C programming language. Dynamic C also allows for programs to be written which endow the robot with total autonomous control. Figure 1-1 illustrates the above mentioned components.

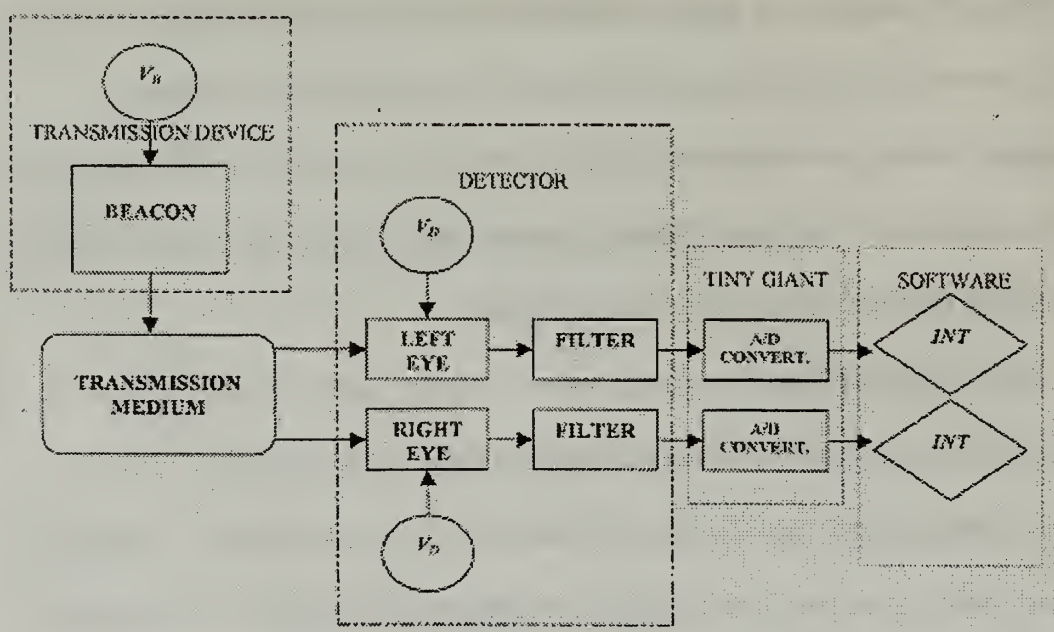

Figure 1-1 . Schematic of optical detection system used on robots for SE 3015 Robot Wars competFition.

This thesis builds upon previous work, [Jones, 1997], in providing a foundation for the accurate simulation of a two-robot competition. The goal for the final model is to

serve as a viable aid for students in the design, testing, and evaluation of the robot during the entire systems engineering process. The previous work done in this area laid an initial foundation for a computer simulated model depicting all physical behaviors of the robot due to both external stimuli and internal control albeit in a somewhat rudimentary fashion, focusing on visualization and data recording. This thesis focuses specifically on a more detailed software model of the optical detection elements in addition to the basic control behaviors of actual autonomous robots used in the SE 3015 course.

B. MOTIVATION

The primary motivation for this study arises out of the author's interest in the graphical modeling and depiction of autonomous, physical systems. This interest stems from introductions to the Combat Systems Robot, NPS Autonomous Underwater Vehicle, and various other simulation projects undertaken during several courses in physically based modeling.

Additional driving forces behind my involvement in this area of research stem from lessons learned during the testing and evaluation phases of engineering the Combat Systems Robot. During these processes, questions always arose as to how robot design and testing might be made easier. My response always seemed to entail some graphical model, which enabled students to actually design and test their robot either alone or against another robot. By performing tests of this nature, the student would be better equipped to visualize the projected performance that the robot was being designed to achieve.

C. PURPOSE

The purpose of this thesis is first and foremost to replicate the performance of the optical transmission and detection components of the SE 3015 Combat Systems Robot. To accomplish this, a transfer function approach has been taken in order to represent each of the components and effects involved step-by-step, from the initial beacon light emission to the final analog-to-digital (A/D) conversion. Each of these processes has been assessed and the input to output transfer function developed. The transfer function approach yields a modular system, for modeling, simulation, and experimentation, where modules can be independently modified if robot elements change.

Given that the primary purpose of each component in the detection sequence has been assessed, the secondary purpose of the thesis is to determine which of the components are required to be used within the simulation and to what extent they must be replicated. Due to the fact that the detection sequence lies at the very heart of the direct application of this autonomous robot, such determinations go a long way in aiding future verification and validation of the robot models with respect to the actual, physical robot.

D. THESIS ORGANIZATION

This thesis consists of detailed physical descriptions and characteristics of the optical beacon, detection circuitry, and the photodiodes. It then describes the measured behavior of all the above components and defines an input-to-output transfer function for each. Simulation models for each of the components are provided. Component models are then composed to produce and integrated simulation for total virtual control, detection, and tracking of the autonomous robot.

Chapter II provides details of previous robot wars simulation work. Chapter III gives a detailed description of the simulation problem addressed in this thesis. Chapters IV and V provide physical descriptions of the transmission and detection components of the system, including their theoretical and measured behavior. These two chapters also define the input-to-output signal transfer function associated with each component and describe how the component models are simulated. Chapter VI gives a synopsis of the transfer functions and a step-by-step process of how they are used. Chapter VII describes how each of the defined transfer functions is linked for comprehensive simulation of detection and tracking of a robot. This chapter also discusses the software and code used to simulate the internal and external behavior of the robot. Chapter VIII discusses how each of the transfer functions was tested and how the overall simulation was tested for overall performance. Chapter IX describes the simulation software, its components, and how to use it. Chapter X presents conclusions and future work, emphasizing and extending the simulation for robot testing and evaluation modeling in the SE 3015 Robot Wars.

II. PREVIOUS WORK

A. INTRODUCTION

The majority of the initial work done in simulation of the SE3015 Combat Systems Robot (Simbot) is described in *Robot Wars Simulation* [Jones, 1997], a thesis which lays the foundation for each of the elements needed for the final model. In that work, Jones describes, in detail, the physical makeup of the robot, its detection sequence and the rules and boundaries of the actual Robot Wars Competition. In addition to a description of the robot components, an object-oriented approach is used to outline and characterize each of these components in the C++ programming language. The specific robot components on which the work focuses are the robot base, its optical detection system, and its standard weapons.

For the robot base, its static equations of motion were derived and the commands which allow for control of the real robot were "planted" within the Simbot so that it behaves in the same manner as its real counterpart. For the optical detection system, a basic set of equations was used to compute the final digital, integer number produced by the A/D converter and used by software to control the robot. In that work, the initial integer number was computed based on the range and angle from the beacon to the robot. For the robot's standard weaponry, basic, static based equations of motion were used to compute projectile motion of the standard ammunition from the robot to its target. Other auxiliary features that were implemented include robot collision detection and playing-field edge detection. Each of these components was programmed within a single simulation, which implemented and gave each the interconnectivity required for

depiction of robot detection and acquisition sequences. That thesis concludes with a rudimentary, graphical, OPEN GL based depiction of the robot's engaged in the sequences performed in a Robot Wars Competition.

B. PRESENT WORK

For this thesis, initial work began in the CS 4472, Physically Based Modeling, class. In this class the author and two other Combat Systems students who had completed the SE 3015 course derived the dynamic-based equations of motion for the robot. Although the accelerations and decelerations values are theoretical, this allows for future students working in this area to actually compute these values based on realistic robot behavior and directly input them into the program which computes the positional values of the Simbot based on given commands. Additional work in the area was conducted during the authors course work in the CS 4202, Introduction to Graphics, class. In this class the author improved upon the previous graphical depiction of the Simbot and its features using the Virtual Reality Modeling Language (VRML) 2.0 as the graphical tool. The positional and angular data computed by the earlier program were used within the VRML program as an engine for the Simbot to move within its "playing field" world.

In this thesis, the foundation for the optical detection system previously created is improved by including detailed modeling for each system component. This system model is interfaced with the author's work using Jones' program connectivity as a guide, for the purpose of graphically visualizing Simbots engaged in various optical detection sequences.

C. SUMMARY

In summary, this chapter has briefly discussed the foundation laid in the area of modeling and simulation of the SE 3015 robots as well as the Robot Wars competition. In addition, previous work done in this area by the author was discussed along with the emphasis of this thesis.

III. PROBLEM DESCRIPTION

A. INTRODUCTION

This section addresses the specific component of the SE 3015 robot of which this thesis focuses and the coinciding simulation problem.

B. SCOPE

The scope of this thesis is to accurately simulate the search, detect, and tracking functions of the autonomous robot used in the SE 3015 class and Robot Wars Competition. This work greatly enhances the foundational simulation previously developed by improving the physics for a specific component of the Simbot, the optical detection system. This thesis also refines the graphical or virtual depiction of the Robot Wars Competition by using the VRML 2.0 3-dimensional (3D) graphics tool. The use of VRML also allows for immediate Internet access to the simulation as well as the possibility of future networking. The goal of the simulation is to enable one to compare and analyze the design, experimental and simulated behavior of the robot, both internally and in reaction to an external stimulus.

C. DETECTION AND TRACKING SEQUENCE

The initiation of a detection and tracking sequence begins with an LED beacon emitting light at an assigned frequency, which is detected by two photodiodes or "eyes" of a robot. The diode, in turn, produces current, which is the following circuit converts into a voltage and filters in preparation for input to the Tiny Giant's A/D converter. The

Tiny Giant then converts the analog data into a digital number to be used in robot control software. Because the inputs and outputs of the various sub-components of the optical detection system are vital to robot control, it is essential that the simulated components provide a high degree of realism. To achieve this, the determination is made as to what components must be modeled and to what fidelity each of them is simulated in order to provide overall simulation accuracy, while not having excessively long computations. Once the required components were identified, a signal-transfer function approach was used to computationally simulate an initially transmitted signal, its detection, and all subsequent processing to yield the final digital number.

As previously stated, the robot receives optical signals with two photodiode "eyes". The robot then processes the signals, converts them to digital form, and finally interprets the magnitudes and differences to determine the location of the opponent. The controller drives the robot's response. The processor, interpreter, and controller logics and software implementations are designed by the student. Figure 3-1 illustrates this concept.

D. SIMULATION PERFORMANCE

The simulated robot, the Simbot, operates in somewhat the same manner. However, in the simulation there are no real objects or signals, and the position of the opponent is known. Thus the simulation must replicate the performance of the beacon and the resulting light intensities received by the eyes due to beacon distance, opponent's angular location, and detector eye geometry. The simulation does not modify a signal flowing through successive circuits as the real system does. In contrast, it applies a set of

transfer functions to the end-to-end, beacon to A/D result. The final information "passed" to the simulation interpreter is the same as the robot A/D signals. The simulation must then replicate control commands as designed for the robot.

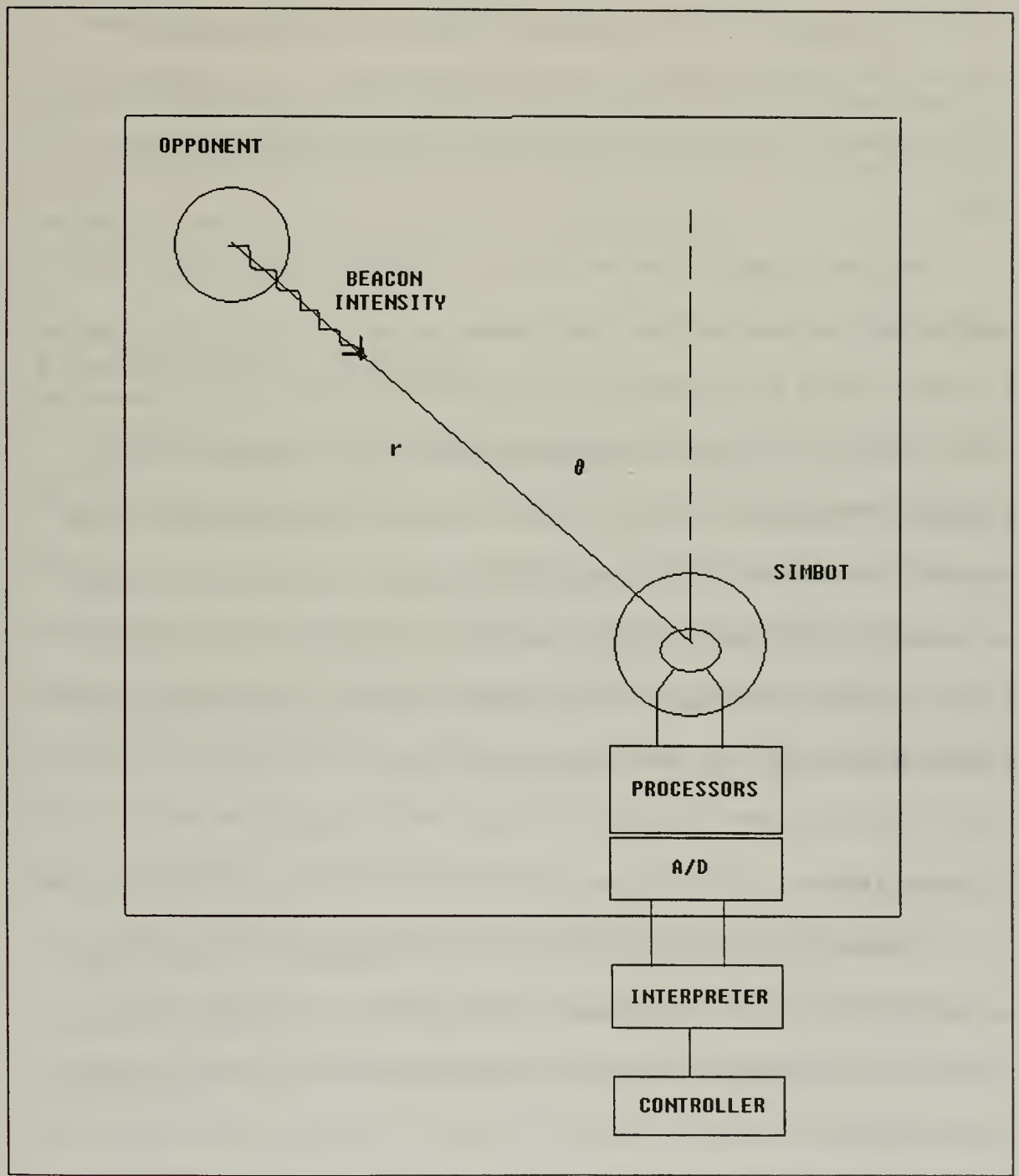

Figure 3-1. Diagram of simulated detection sequence. The Simbot detects the light trasmitted from the opponent's simulated beacon. The simulation deals only with the boxed area.

As with all high-level programming, an abstraction or black box approach is taken where the user is intentionally shielded from the details or inner workings of the simulated model. Because of this approach, the user is not sure how complex the simulation is, but is aware of how effective the model is in representing real, physical behavior. The problem at hand is to determine how accurately each of the components must be modeled, both individually and collectively, so that they adequately depict reality.

Each robot is assigned a beacon frequency for which an electrical circuit is to be designed, built, and tuned to detect. During testing, evaluation, and the final competition the beacon is placed at some arbitrary position level with the robot's "eyes" or on another robot's platform. The robot is then activated for an autonomous sequence of searching, detection, and tracking of the beacon. The simulation seeks to keep the integrity of this sequence of events intact, offering a graphical interpretation of the process while doing so. In addition to the detection sequence, the actual algorithms and programs used to control the robot are also simulated so that the Simbot reacts to its external stimulus in the same manner as the actual robot is programmed to do.

E. SUMMARY

In summary, this thesis accurately models the LED beacon, robot "eyes", and attached electrical circuitry using student robot input parameters, but does not modify any other Simbot methodology, thus leaving the remainder of Jones' baseline methodology as originally designed.

IV. ROBOT BEACON

A. INTRODUCTION

This chapter discusses the transmission component of the optical detection system. It addresses the beacon's physical characteristics and both its theoretical and measured behavior. The chapter concludes by defining the transfer functions for each of the pertinent transmission elements required for the simulation.

B. PHYSICAL CHARATERISTICS

From the system schematic illustrated in Figure 1-1, it can be seen that the robot beacon is the first component in the optical detection sequence, being the transmitter which the detector is tuned to acquire. The LED beacon placed on each robot is the specific object that is detected by its opponent. Each beacon is set to flash at a unique frequency generally within the 5-25 kHz range. The frequencies of each beacon are separated by a multiplication factor of 1.22, with the lowest beacon frequency assigned, 5 kHz. The frequency separation allows most of the unwanted harmonics from other robot beacons to be rejected. A CD4047 multivibrator integrated circuit driven by a nominal 6-volt supply is used to produce the square wave, which drives the beacon's flash.

The beacon has a total height of 2.25" and a radius of 0.56". For detection purposes, the beacon is placed underneath the center of a wooden disk located 3.5"- 4.5" above the top of the Tiny Giant. The beacon's light distribution pattern is 360° (i.e. omni-directional) in the horizontal plane. In the vertical plane the beacon's light distribution pattern is ±5° about the horizontal plane, for a total of a 10° vertical angular

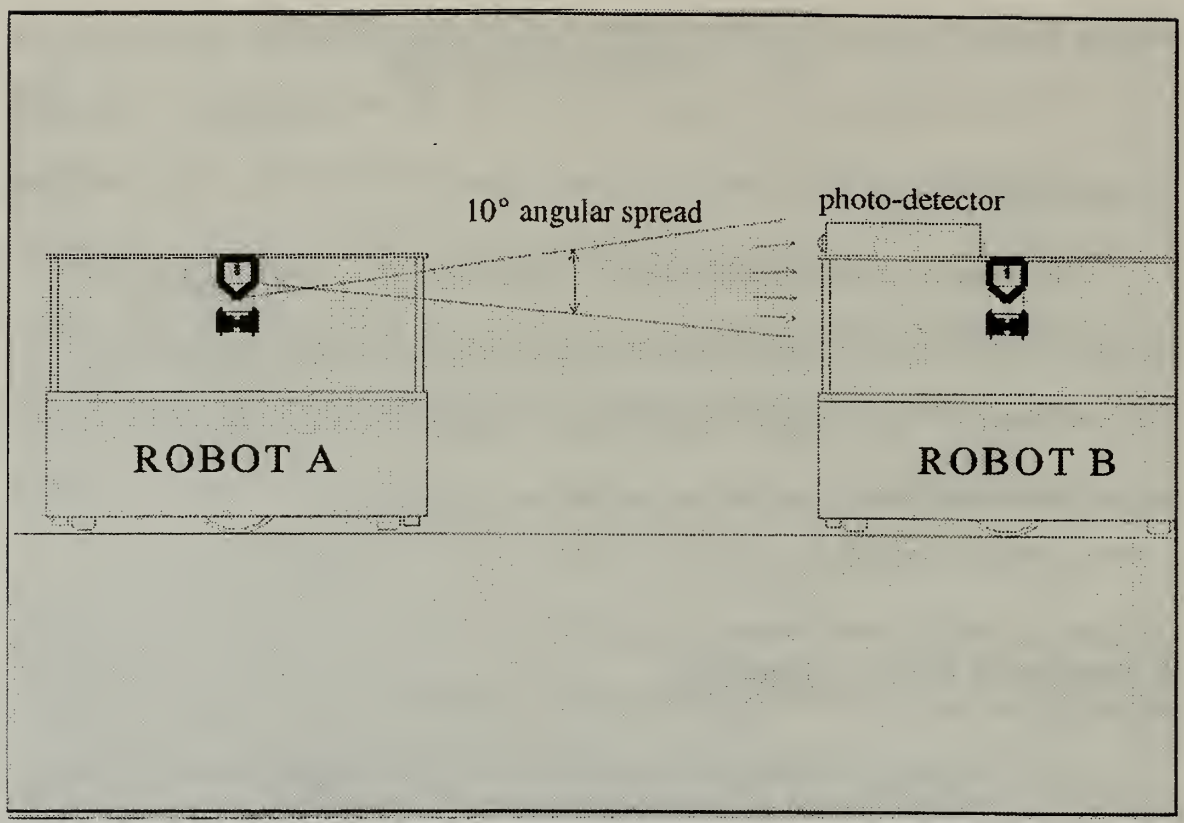

Figure 4-1. Beacon positioning. The beacon is positioned underneath the top platform of the robot's base so as to give a 10° total vertical spread.

spread. Figure 4-1 shows how the beacon is positioned on the robot. The only physical entities on the robot which obstruct the view of the beacon are the relatively thin cylindrical platform supports which are used to attach the wooden disk to the top of the Tiny Giant.

C. THEORETICAL BEHAVIOR

The light emitted from the beacon may be compared to light generated from an isotropic medium from which light is distributed uniformly in all directions within the horizontal plane. The transmission medium through which the light is propagated is air. Because the pattern generated by beacon light distribution is spherical in nature, the

beacon is taken to be a simple source. The theoretical intensity of the beacon light as a function of distance is

$$I_T \propto \frac{I_B}{r^2}, \qquad (4-1)$$

where $\mathbf{I_B}$ is the source intensity of the beacon. The factor $1/r^2$ in the equation represents the decrease in received intensity with distance from the source.

In addition to the behavior of beacon intensity as a function of range, another beacon characteristic that must be taken into consideration is the frequency at which the beacon oscillates and its dependence on the beacon oscillator supply voltage. In general, there are two supply voltage options that may be used to power the oscillator. The first is a 9-volt battery and the second is a 6-volt battery pack composed of four, 1.5-volt AA batteries. For each of these options some drainage of the battery capacity occurs over time. As a result of decreasing battery voltage, there is expected to be a subsequent decrease in the frequency at which the beacon flashes. A corresponding decrease in the intensity of the beacon is also expected.

D. MEASURED BEHAVIOR

For the purposes of the ensuing simulation, each of the above mentioned physical characteristics was measured to provide quantitative understanding of the degree of emphasis necessary for accurate depiction. The results of the quantitative measurements were used to derive a transfer function, which relates the pertinent dependencies.

First, a photometer was used to attain a relative measure of the received intensity generated by the beacon as a function of the distance from the source. Two sets of data were taken. The first set of photometer readings was taken in a relatively dark environment. This was done in an attempt to separate the light generated by the beacon from all other ambient and external light sources. The beacon acted as the source of light, and the photometer acted as the receiver. The photometer was placed in a fixed position relative to the beacon. The beacon was then moved away at ranges varying from a minimum of 1" to a maximum of 20', which approximates the maximum range of

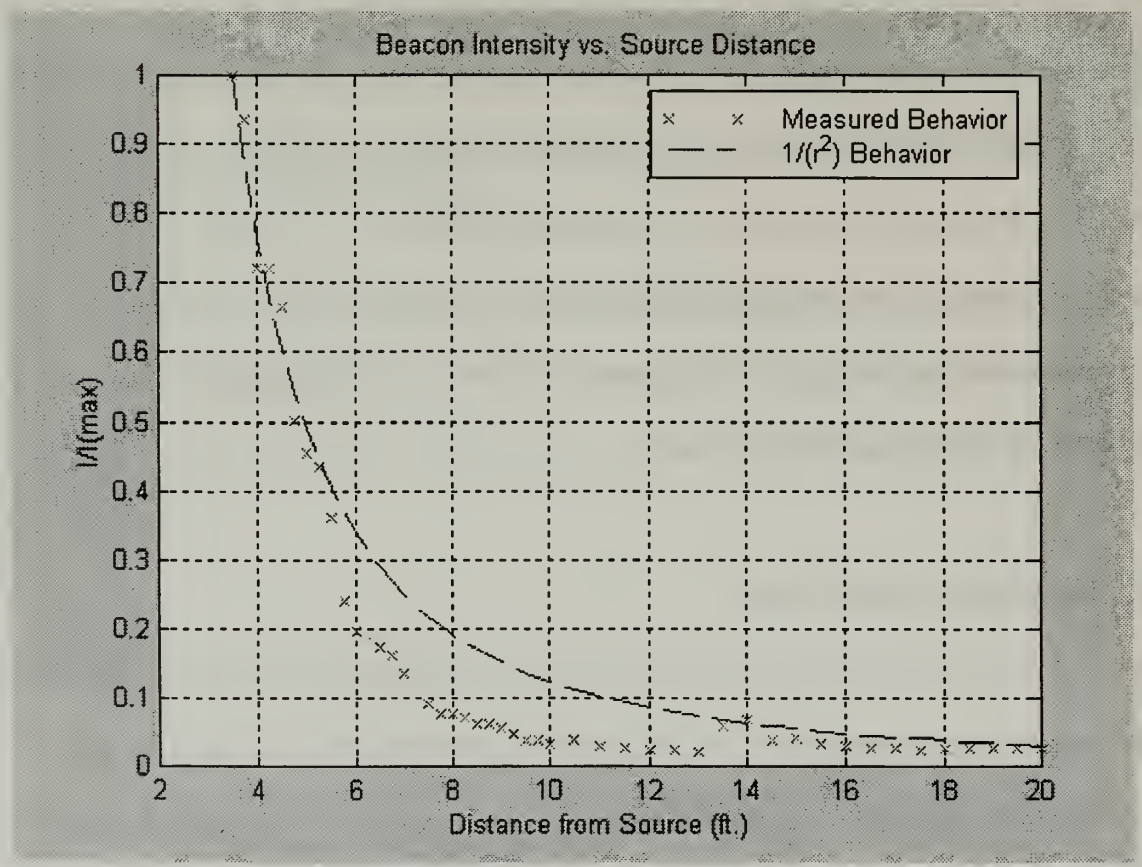

Figure 4-2. Relative photometer intensity readings vs. range. The experiment was conducted under conditions of minimum or no background lighting. The graph shows data for distances after the maximum saturation range.

detection during a robot competition detection sequence. Relative intensity measurements were taken at 6" intervals.

Figure 4-2 shows the results of the experiment. One of the pertinent features of this graph is that the normalized intensity ratio maintains its maximum value of 1 out to a distance of 3.5 feet. This infers that the photometer was saturated for shorter ranges.

In order to compare the intensity distribution as a function of range with the theoretical $1/r^2$ behavior, the data from the curve in Figure 4-2 was analyzed from the point where saturation of the photometer was no longer present. As can be seen from the figure, the measured data bears close resemblance to the expected $1/r^2$ behavior. As a result, for the purposes of the simulation, the intensity as a function of distance can be modeled as $1/r^2$.

The next set of experiments was designed to determine the effects of a second dependency of the beacon transmission, the flash frequency. Due to the fact that a supply voltage drives the beacon oscillator, a determination of the voltages that provide intensity as a function of range had to be made. In addition to these performance characteristics, the affects of the supply voltage on the frequency and intensity had to be measured.

The first experiment measured the beacon's intensity as a function of the supply voltage. In this experiment, a photometer was used to determine the relative intensity of the beacon as the beacon's power source was decreased. Measurements were taken at ranges of 5 and 10 feet. Figure 4-3 shows the results of this experiment. As can be seen from the figure, at supply voltages greater than 4.5 volts, the intensity of the beacon remained constant. However, as the voltage was decreased to below 4.5 volts there was a

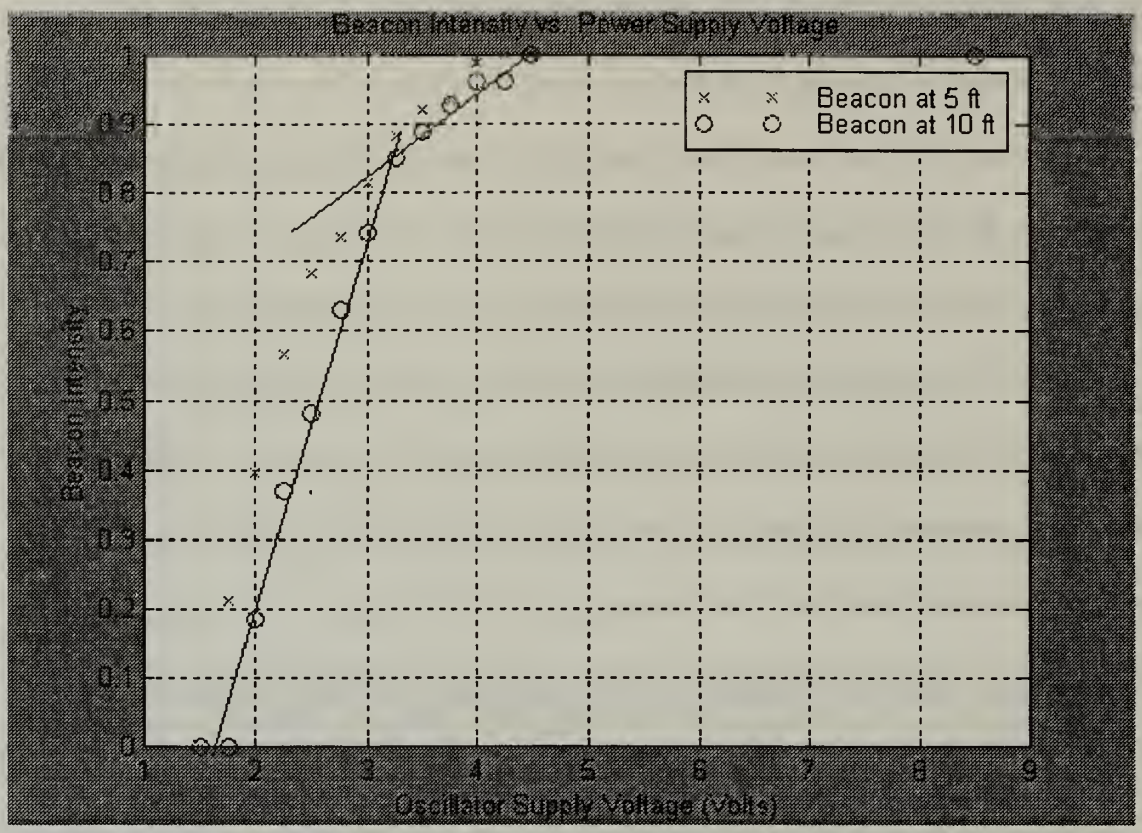

Figure 4-3. Relative Beacon intensity as a function of supply voltage. Slopes for the two linear regions shown define intensity behavior when supply voltage is less than 4.5 volts.

steady drop of intensity. This drop persisted until the light from the beacon was totally diminished at about 1.6 volts. With either of the battery configurations used with the beacon, there is some degradation of the battery output voltage with time. Analysis of the data revealed that for voltages below 4.5 volts, the normalized intensity fall off may approximated in two linear regions as indicated in the figure. For voltages above 4.5 volts there is no change in normalized intensity. Table 4-2 shows the slopes and their corresponding regions.

The next experiment in this set measured the voltage drop of a 6-volt battery pack used to power the beacon oscillator as a function of time. This was done to determine the relative decrease in voltage over a time interval of several minutes due to loading from

$\Delta I / \Delta V$	*Voltage Range*
0.0	9.0 volts $\geq V_0 \geq$ 4.5 volts
0.133 volts^{-1}	4.5 volts $> V_0 \geq$ 3.0 volts
0.533 volts^{-1}	3.0 volts $> V_0 >$ 1.5 volts

Table 4-1. Slopes and voltage ranges for data in Figure 4-4.

the beacon oscillator. Due to the fact that the time period for single round of Robot Competition is 3 minutes, the experiment was conducted over a 15-minute time intervai to measure beacon operation. In this experiment, four brand-new, 1.5-volt AA batteries were used in the battery pack.

A multimeter was used to measure the oscillator battery voltage and a stopwatch to record the progressing time. The battery pack had an initial voltage of 6.68 volts. Voltage readings were taken on the battery pack every 30 seconds up to 6 minutes and then every 1-minute up to a maximum time of 15 minutes. As can be seen from the figure, the voltage dropped quite rapidly during the first minute, going from its initial value of 6.68 volts to 6.32 volts. During the next four minutes there was a fairly steady drop in voltage from 6.32 volts to 5.99 volts. The next eleven minutes had a relatively slow voltage drop from 5.99 volts to 5.66 volts.

At each of the time intervals, the frequency resulting from the decrease in voltage was also measured. The frequency's dependence on supply voltage is shown in Figure 4-5. As can be seen from the figure, the frequency decreases with decreasing voltage. For the simulation, the frequency decrease is approximated as being linear. The slope is used to calculate the decrease in frequency as a result of the oscillator battery voltage decrease. The slope is given by

$$\frac{\Delta f}{\Delta V} = 23.148 \text{ Hz per volt.} \qquad (4 - 2)$$

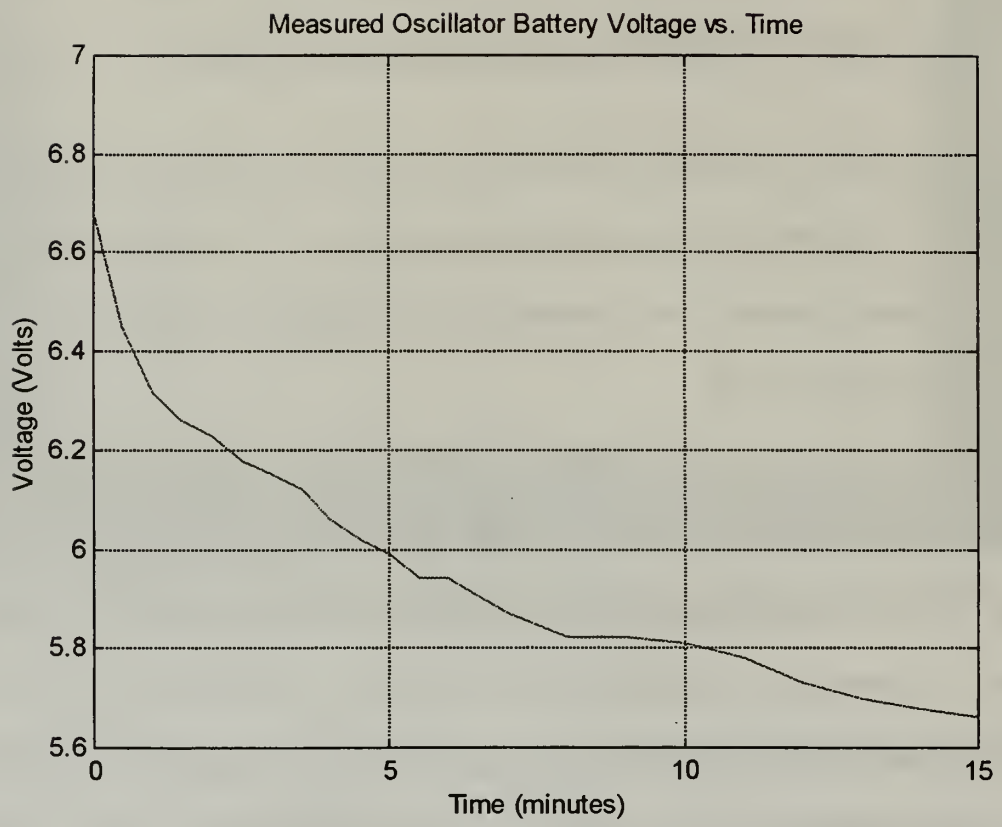

Figure 4-4. Oscillator battery voltage as a function of time. Test conducted using four 1.5 volt AA batteries, under minimum loading conditions.

The decrease in beacon oscillator battery supply voltage over time and resulting oscillator frequency decrease for any detection sequence is largely dependent on battery loading (i.e. the amount of current supplied by the battery). For the data shown in Figures 4-4 and 4-5, the current was not measured. A battery pack consisting of four 1.5 volt AA batteries was used as the oscillator supply voltage. The beacon oscillator was the only load placed on the battery pack. For the simulation, the battery drainage over

time was approximated using straight-line slopes for the linear portion of curves of commercial 9-volt batteries, under minimal loading conditions of 20 mA, for time periods under two hours.

For the purpose of the simulation, the user is allowed to select from four different loading conditions, heavy (H), medium (M), light (L), and no-load (N), for batteries used

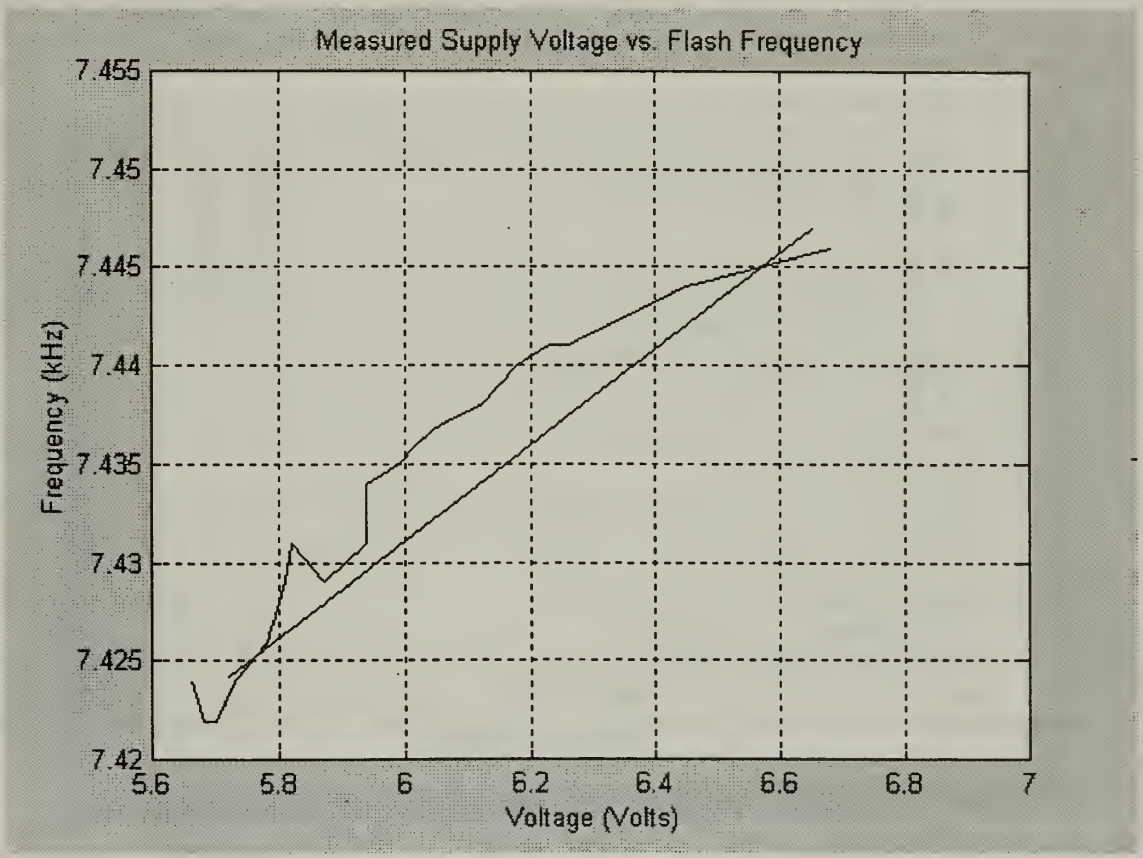

Figure 4-5. Beacon oscillator frequency as a function of supply voltage. The frequency of the oscillator decreases with supply voltage. For the simulation, the decrease was taken to be linear. The linear region used was from 6.6 to 5.8 volts. Minim battery loading is assumed.

with the Simbot. The default condition is the light, 20-mA case. Each of the loading conditions has a different linear, time-dependent decrease in voltage and serves as an

approximation of various loading levels. The selection of an increased loading condition results in a faster rate of decrease in supply voltage with time. This faster rate of decrease ultimately effects the rate at which the oscillator frequency decreases. If more accurate battery voltage discharge and loading characteristics are desired, the current levels should be measured and compared to voltage discharge curves for the specific type battery used.

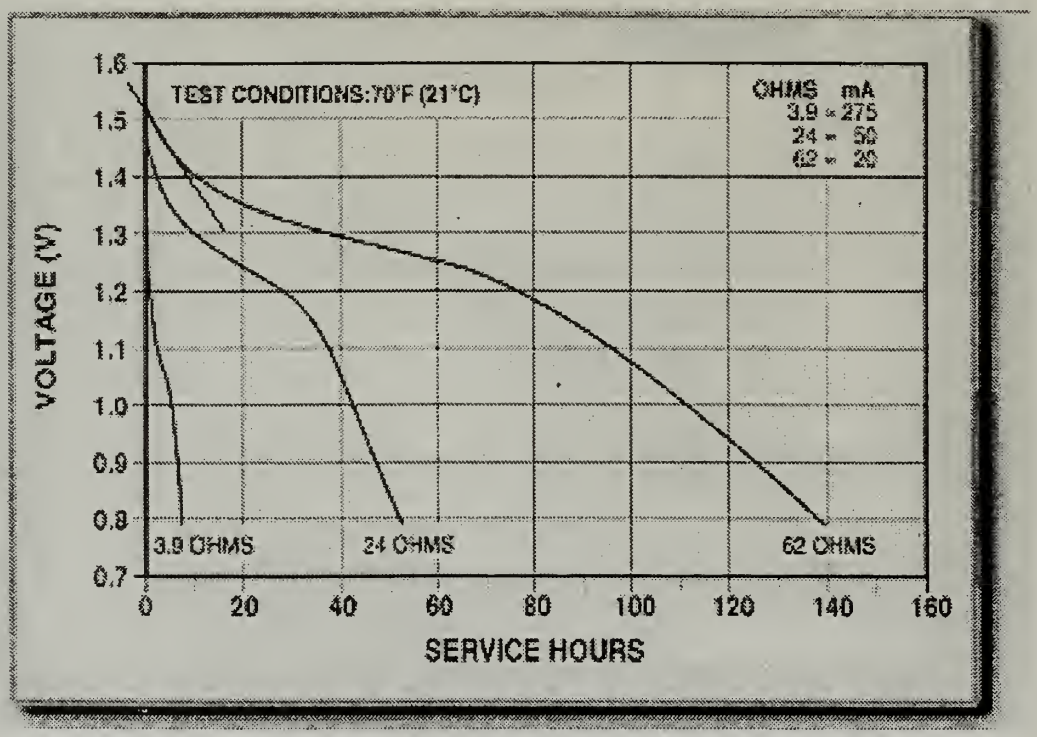

Figure 4-6. Voltage discharge characteristics for a commercial 1.5-volt battery. The 3 loading condition discharge curves shown are used to simulate heavy, medium, and light loading conditions of the Simbot detector.

The simulation uses the slopes of linear regions for battery voltage discharge curves similar to that shown in Figure 4-6. The matrix in Table 4-1 shows the allowed

user selections for loading conditions along with slopes used for each condition. Each of the values listed in the table are given in the scenario as placeholders so that they may changed if other measurements are made or other batteries are used. The curves for a 9-volt commercial battery are used in the simulation as the default battery type.

Since the actual beacon used in the Robot Wars competition always has its own battery supply, whose only load is the beacon oscillator, the light-loading condition is used to simulate it. The slope of the line for conditions of light loading of a 9-volt battery has a value of 2.53×10^{-6} volts per second. This value is used to determine the change in simulated beacon oscillator voltage with elapsed Simbot operation time. The

BATTERY TYPE	**LOADING CONDITION**		
	H	**M**	**L**
9-volt battery	1.04×10^{-4}	5.56×10^{-5}	2.53×10^{-5}
	H	**M**	**L**
1.5-volt battery	5.56×10^{-5}	5.56×10^{-6}	2.78×10^{-6}

Table 4-2. Matrix of battery types and loading conditions used in simulation. The values listed below the loading conditions are the slopes used. The slopes are given in units of volts per second.

The linear portion used is shown, in Figure 4-6, as a line imposed on the 20-mA curve for service hours less than 10.

E. CONCLUSIONS AND RESULTING TRANSFER FUNCTIONS

From analysis of the results of each of the conducted experiments, it may be concluded that the intensity transmitted by the beacon may be characterized by

$$I_B = I_{max} \cdot F_I(V_B), \qquad (4-3)$$

where I_{max} is the maximum beacon intensity possible under conditions of maximum supply voltage and I_B is the transmitted beacon intensity, which is dependent on V_B, the beacon supply voltage. F_I is the related transfer function, given by

$$F_I = 1.0 - \Delta I, \qquad (4-4)$$

where ΔI is calculated by multiplying the beacon battery supply change in voltage by the appropriate slope given in Table 4-2.

In addition, the beacon flash frequency is given by

$$f_B = f_o - F_f(V_B), \qquad (4-5)$$

where f_o is the frequency at which the beacon oscillator is initially tuned at the beginning of a detection sequence and f_B is the transmitted beacon frequency, which is dependent on the beacon supply voltage. F_f is the related transfer function, given by

$$F_f = \Delta f, \qquad (4-6)$$

where Δf is calculated by multiplying the change in beacon oscillator voltage by the slope given in Equation 4-2.

As stated previously, each robot is assigned a unique frequency, which the detector is designed to acquire. The results of the experiment conducted with and without background lighting shows the reason for needing to include the beacon oscillator's frequency dependence. As seen in Figure 4-2, when no background lighting was utilized, the beacon was easily seen out to the maximum distance of 20 feet. Additional tests revealed that when background lighting was present, the beacon became

non-distinguishable beyond about 5 feet. The distinct frequency of the beacon allows the detector to easily acquire the beacon in an environment which may consist of other emitting beacons and additional background lighting. A small change in beacon frequency can seriously affect robot performance.

From the results of the beacon intensity as a function of range experiments, it may be concluded that the beacon intensity range dependence may be approximated as $1/r^2$ at ranges between the maximum saturation distance and the maximum detection ranges. For the simulation, the user will be asked to supply these two ranges. The normalized intensity at ranges less than the maximum saturation distance is 1, while the intensities at ranges greater than the maximum detection range are given as 0. All subsequent intensities for a given range will be calculated by computing the normalized intensity from $1/r^2$.

This chapter shows the results of several experiments used to determine the characteristics of the transmission beacon required. It has been concluded that beacon intensity and frequency must be simulated for adequate replication of the optical detection sequence of the SE 3015 Combat Systems Robot. Furthermore, the beacon intensity has been determined to be a function of its flash frequency and oscillator supply voltage. In order to do this, the time dependence of the oscillator supply voltage must be simulated.

Finally, in order to pass the beacon intensity on to the other components in the sequence, in the simulation, the following steps are used. First of all, the simulation is set up using user input initial values for V_B, f_0, and I_{max} at $t = 0.0$, where t is the scenario time. Additional user input parameters include battery type and loading condition. Using

these inputs, the loading condition selects the appropriate slope for the battery voltage degradation with time. Then, given some $t > 0.0$, the resulting beacon frequency is calculating from Equation (4-3) and the intensity calculated from Equation (4-5). Procedures for measuring the required input parameters are given in Chapter VII.

One additional dependency on the received intensity is that of the intensity of light present as a result of various background sources. As a result of these additional light sources which produce a cumulative intensity, the final transmitted intensity is given as

$$I_T = I_B + I_N , \qquad (4-7)$$

where I_N is the background or noise intensity, which must be measured by the user and is an input parameter for the simulation.

F. SUMMARY

This chapter discussed the optical beacon component of the optical detection system. It described the beacon's physical characteristics as well as its theoretical and measured behavior. Transfer functions were defined that allow the pertinent signals transmitted from the beacon to the detector to be effectively modeled. As a result of these transfer functions, the simulated beacon transmits two time-dependent and voltage-dependent quantities to the Simbot, the beacon intensity and the beacon frequency.

V. DETECTION SYSTEM

A. INTRODUCTION

This chapter discusses the receiving component of the optical detection system. It addresses the detector's physical characteristics and both its actual and measured behavior. The chapter concludes by defining the transfer functions for each of the pertinent detector elements required for the simulation.

B. OVERVIEW

The detection system receives the light transmitted by the beacon and processes it into a series of voltages needed to produce the final A/D intensity integer. The system's detector consists of two major sub-components, its detectors (the photodiode "eyes"), which actually receive the light and the processor, which does the required signal processing. This chapter describes the physical positioning of the "eyes", explains the behavior of both the signal processing circuit and eyes, and additionally defines a set of signal input-to-output transfer functions, which the simulation uses to transform received intensity into an integer number, emulating the A/D converter.

C. DETECTOR PHYSICAL CHARACTERISTICS

A minimum of two optical diodes must used on each robot to enable it with range and angle discrimination. Each of the diodes is attached to its own signal processing circuit, which enable each "eye" to see. As light is emitted from the beacon and received by the active portion of the diode, current is generated and supplied to the first stage of

the processor. The active portion of the optical diode is rectangular in shape. The horizontal length is 0.28 in and the vertical height is 0.16 in. The total effective active area is 0.045 in^2. The eyes are usually placed on the front of the robot platform with the longest side of the active diode area parallel to the ground. The diodes are usually placed at a height at which the active area may still receive light from another robot given robot distances of less than 1 ft. The nominal positioning of the photodiodes and beacon is illustrated in Figure 4-1. As previously stated, a minimum of two eyes is required for each robot. The two eyes are usually positioned side by side at robot centerline and are usually separated by a nose barrier, which provides angular discrimination. Figure 5-1 shows the eye and nose positioning. As can be seen from the figure, the active portion of

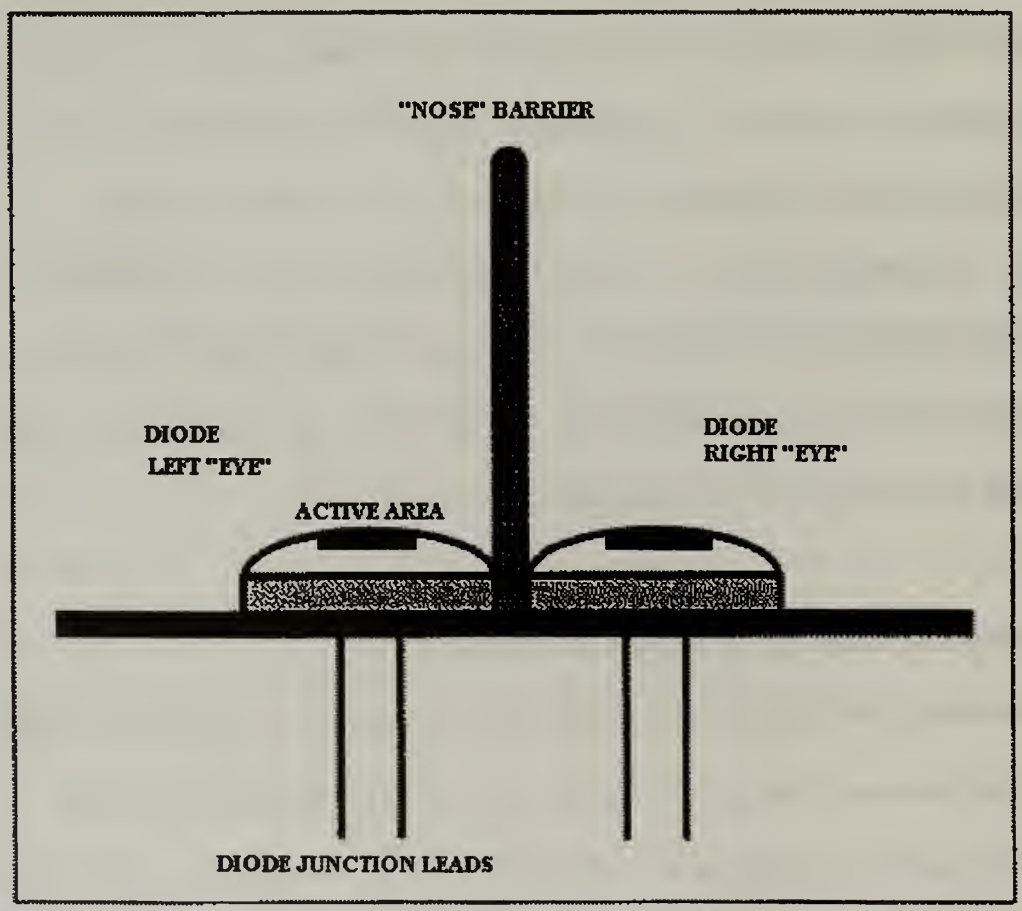

Figure 5-1. Overhead view of typical eye and nose barrier configuration for SE 3015 robot. The photodiode active area and junction leads are also depicted.

the diode lies underneath a glass coating. For the purposes of this thesis, it is assumed that the coating has no lens magnification effects.

D. DETECTION SYSTEM BEHAVIOR

Each eye used on the robot is usually attached to its own processor. Other configurations are possible which use a single processor and a multiplexer to sample the eye output signals. The processor filters and processes the signal received from the eyes in such a manner that the Tiny Giant may use it for A/D conversion. There are three major components in this circuit. The first is a transimpedance amplifier, which converts the current produced by the photodiode into voltage. The transimpedance amplifier is connected to what may be the most important part of the processor, a variable-state bandpass filter, which is tuned to resonate at the assigned beacon flash frequency. This filter has a variable bandwidth that is centered on (and gives maximum response at) the tuned frequency. The last portion of the processor is a half-wave rectifier, the rectified signal being what the Tiny Giant's A/D converter converts into integer numbers upon which command, control, and detection algorithms are based.

The A/D converter converts the analog voltage into an integer number ranging from 16 to 4080. The lower number represents the minimum signal present when there is no detection. When a minimum amount of noise is present, the integer numbers fluctuate between 16 and 32. A minimum beacon detection signal produces a nominal reading of 48, which represents 100% probability of detection in the software detection logic. Ambient light levels which produce a reading greater than 48 will interfere with detection. The high-end reading of 4080 represents the maximum signal strength. The

numbers generated between 48 and 4080 are largely dependent on the range of the beacon from the diode and the angle at which the light is incident on the active portion of a robot eye. Other factors that may impact the readings generated by the A/D converter are the processor gain, the ratio of beacon frequency to tuned-processor frequency, and the beacon intensity. The beacon and detector system supply voltages also affect these parameters.

For the purpose of the simulation, it is believed that the most important part of the processor is the bandpass filter. The typical bandpass filter used by the SE 3015 classes is of the variable state form, shown in Figure 5-2. As can be seen from the figure, four operational amplifiers (op-amps) are used in combination with a number of critical

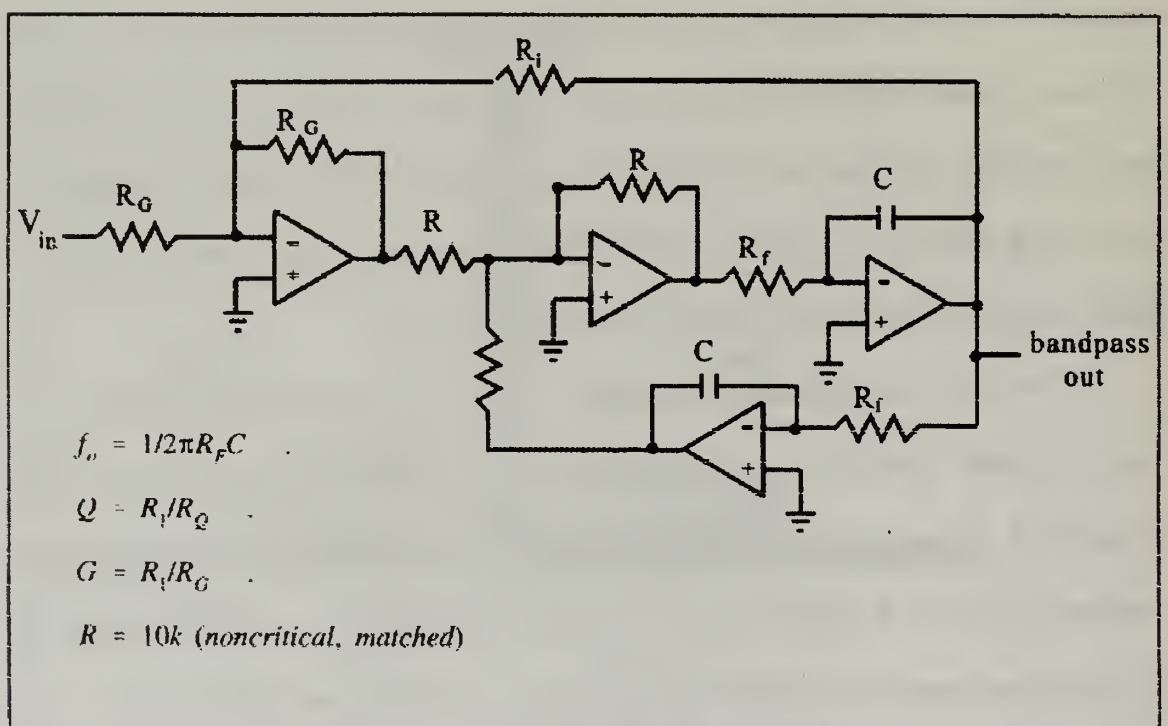

Figure 5-2. Schematic of typical bandpass filter used in SE 3015 course for detection of beacon frequency. This circuit allows resonance frequency, gain, and bandwidth to be set independently of one another.

resistors and capacitors. The advantage of this circuit is that the resonance frequency, bandwidth, and gain may each be set or changed with a single resistor and additionally each setting is independent of the other. As previously stated, the resonance frequency is the assigned beacon frequency. The bandwidth is optional, but is typically designed to give a quality factor equal to 50-100 for optimum performance. The quality factor, Q, is defined as

$$Q = \frac{f_0}{\Delta f}, \qquad (5-1)$$

where f_0 is the resonance frequency and Δf is the bandwidth. The Q gives the relative sharpness of the voltage or current frequency response curve. Figure 5-3 shows

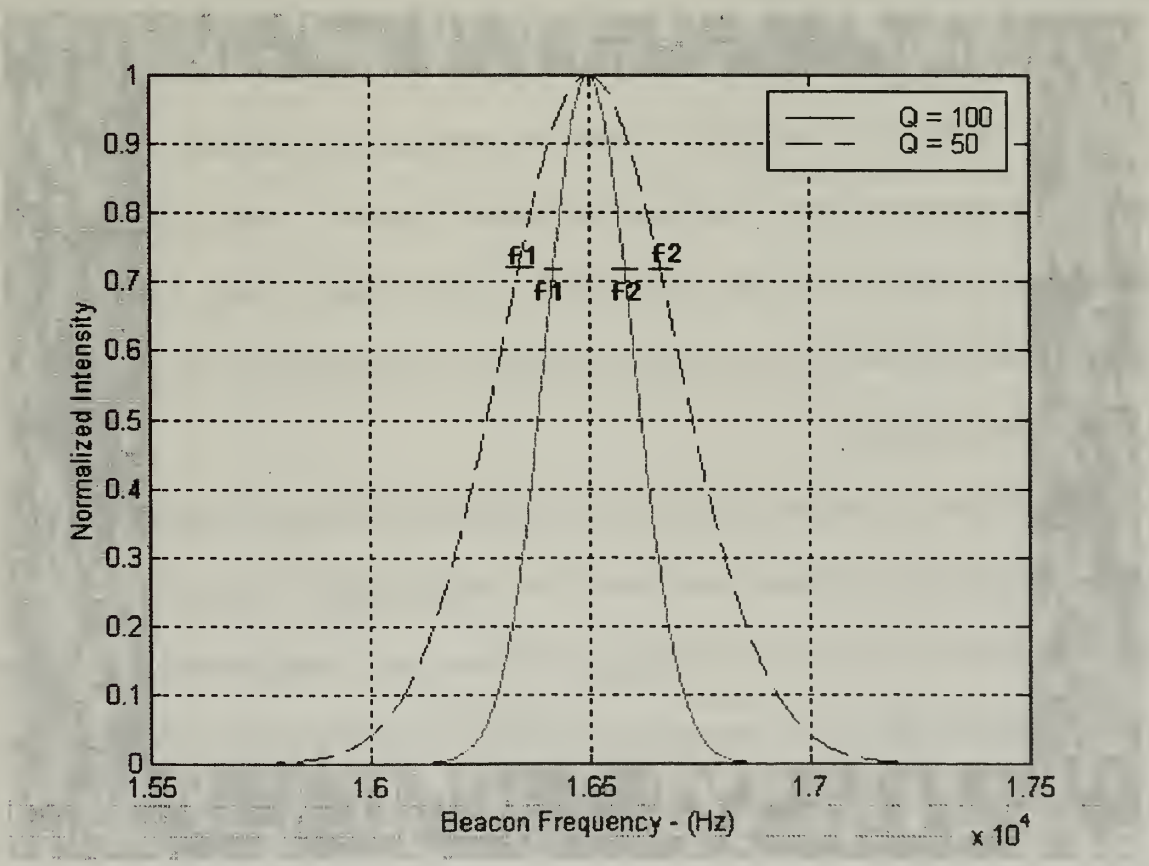

Figure 5-3. Frequency response curve for detector bandpass filter. The figure shows curves and half-power points for curves with Q of 50 and 100.

the measured frequency response of the circuit used for the simulation. As can be seen from the figure, f_1 and f_2 are the frequencies where the voltage drops off by a factor of 0.707 (3 dB down points).

For simulation purposes, the assumption is made that the only internal factor affecting the photodiodes is the processor supply voltage. This voltage will decrease with time due to current drain. The decrease in voltage will cause a subsequent reduction in the performance of the photodiodes. Several experiments were conducted on the photodiodes so that a fairly accurate dependence on circuit supply voltage could be determined.

E. MEASURED BEHAVIOR

For the purposes of the ensuing simulation, each of the detector factors discussed above was measured or analyzed in order to determine the relevancy of each to simulation performance and to develop its transfer function. After designing and operating the processor with the actual robot, it has become evident that the only parameters and data needed for its accurate replication are those dealing with its filter and its voltage gain dependency.

The frequency response of the circuit was measured by using a signal generator to apply a signal of a few millivolts to the input of the bandpass filter. The output of the remainder of the circuit (input to A/D converter) was examined using an oscilloscope. A half-wave rectified waveform was produced as expected. As the frequency of the input signal was increased or decreased from the resonance, the output was observed to ensure that the voltage of the output waveform "rolled off" as predicted. At both half-power

points, the voltage dropped by a factor of 0.707 when the frequency came within ± 0.1 kHz of f_1 or f_2. The same behavior and results were observed for quality factors ranging from 50-100 in other tests. For an assigned frequency of 16.5 kHz and a Q of 50, the bandwidth is 330 Hz, giving f_1 and f_2 of 16.435 kHz and 16.665 kHz respectively.

In order to determine the effects of voltage gain on processor output, a set of three separate experiments was conducted where the gain of the circuit was adjusted to three different levels: 10, 20 and 40. A single photodiode was placed on a stand situated on a smooth plane at a height of 6" from the ground. A beacon was then situated on a similar stand so that the center of its active portion was the same height from the ground as the diode. The beacon was then moved from distances ranging from 6" to 20' at intervals of

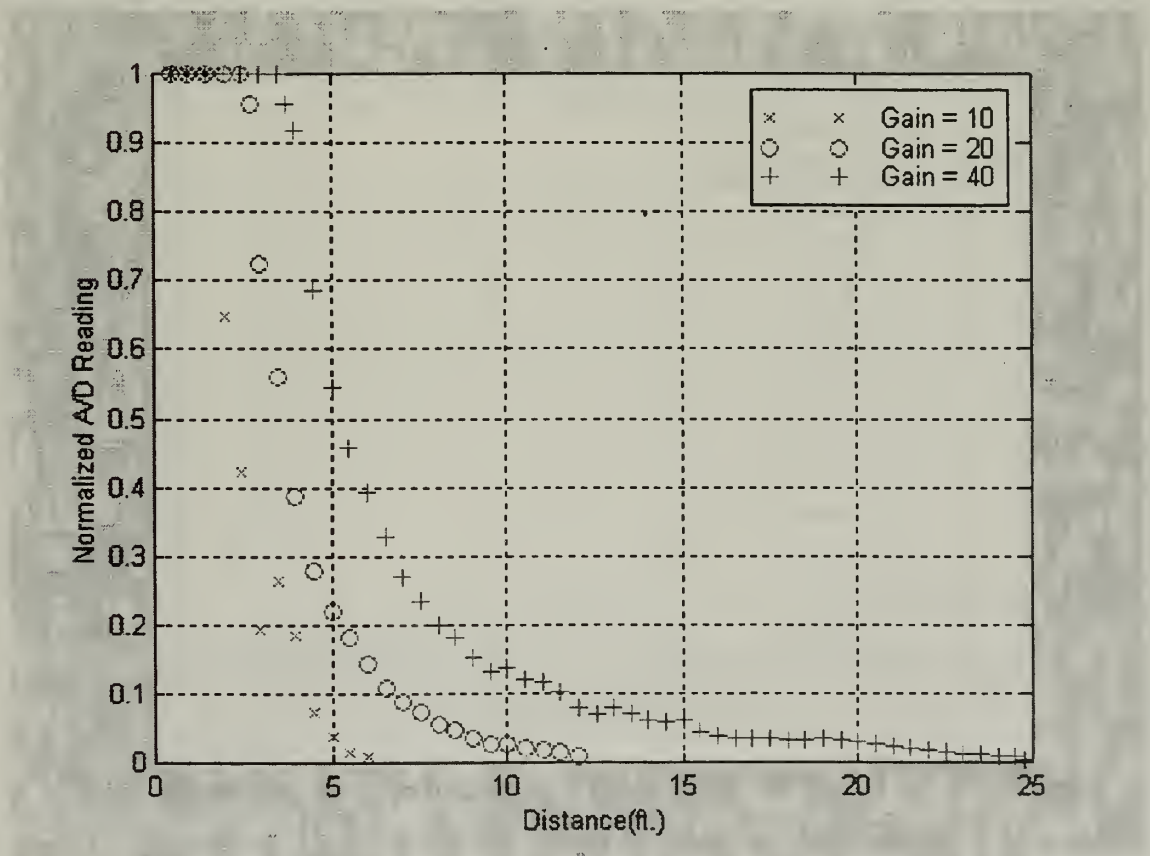

Figure 5-4. Measured A/D reading as a function of range for different gains.

taken at each of the ranges. The results of this set of experiments are shown in Figure 5-4.

From the figure, it can be seen that the gain of the circuit primarily affects two things, the saturation distance (distance at which maximum signal is read) and the maximum range of detection. With a gain of 10, the measured saturation distance was 2.5'and the maximum detection range was 6'. When the gain was doubled to 20, the measured saturation distance increased to 3.2' while the maximum detection range

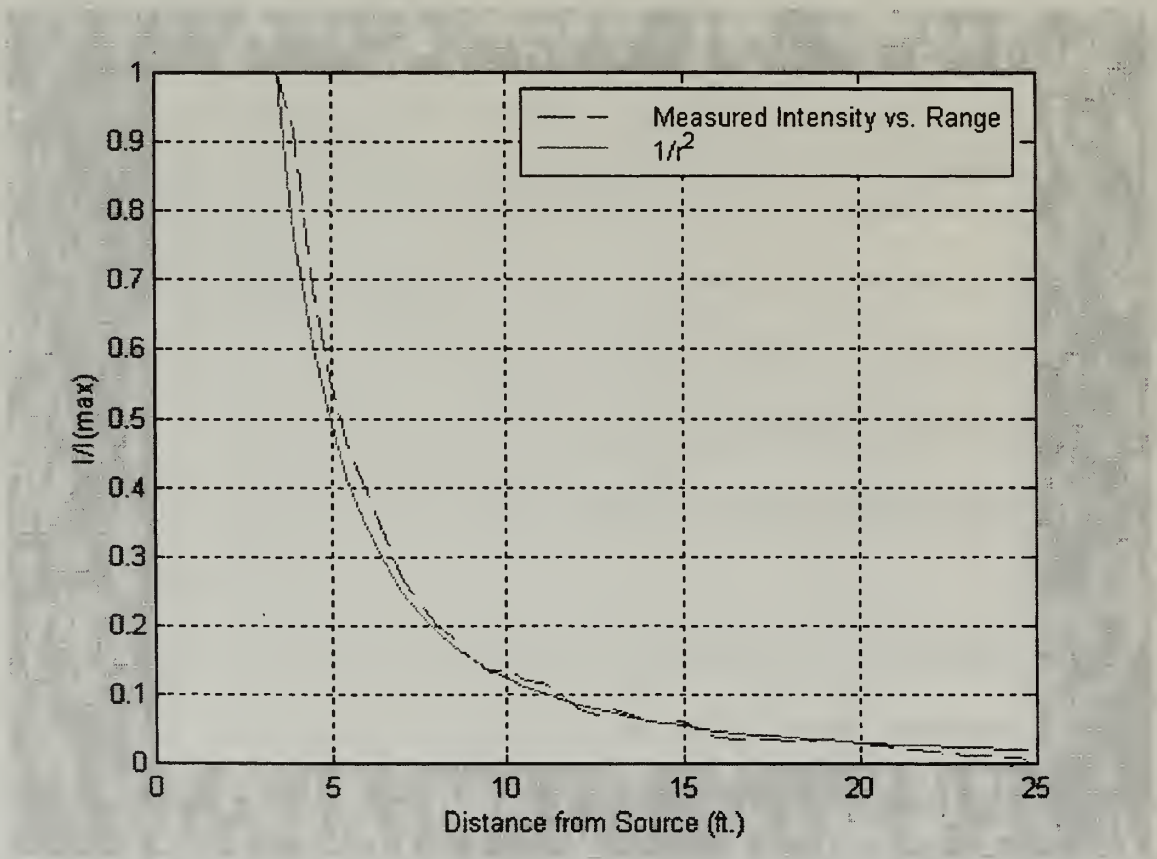

Figure 5-5. Comparison of A/D reading vs. range and expected $1/r^2$ behavior.

increased to about 12'. When the gain was once again doubled, to 40, the measured saturation distance increased to about 4' while the maximum detection range increased to about 20'.

For each of the experiments, the A/D readings versus range curves have a $1/r^2$ behavior from the maximum distance of saturation to the point where the maximum detection signal was measured. This indicates that the detection system is linear, so that an end-to-end transfer function can be used. Figure 5-5 shows the curve for a gain of 40 and the expected $1/r^2$ behavior.

The next experiment involved determining the effect of supply voltage on the detector system. For this experiment, a dual source was used to supply the processor

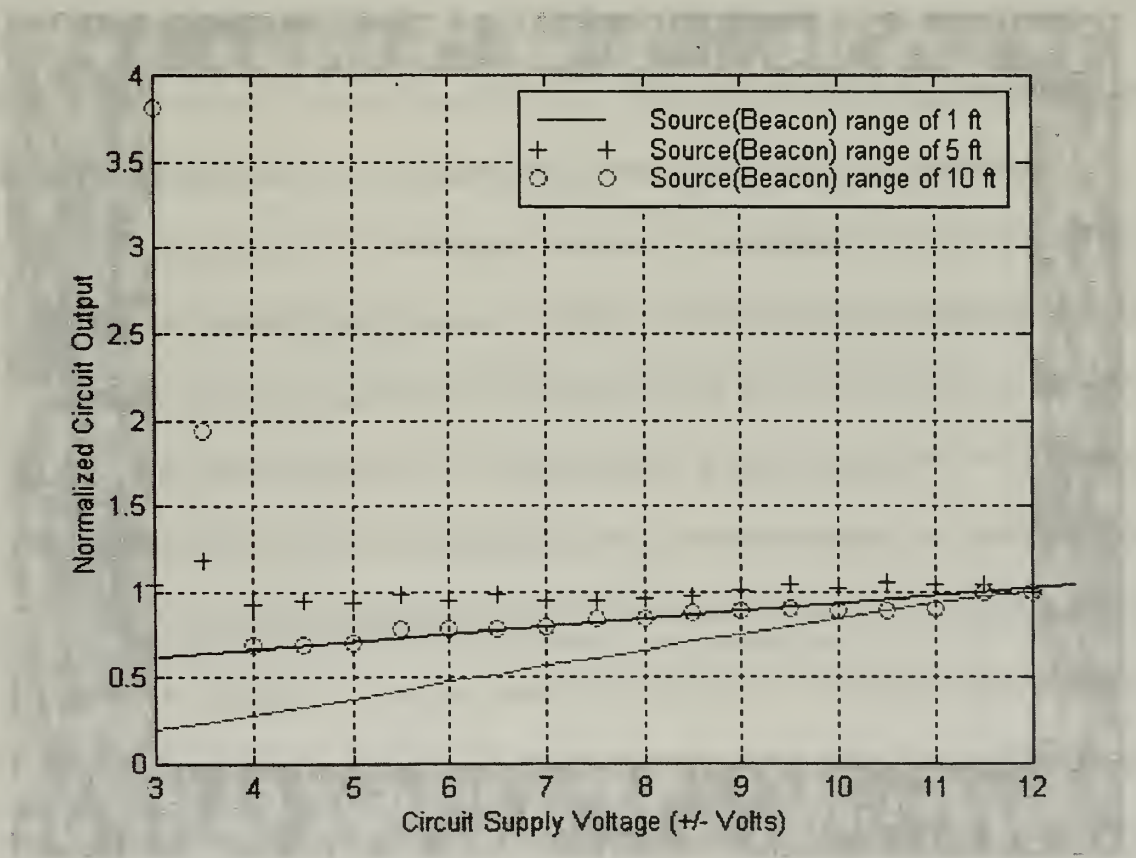

Figure 5-6. Effects of supply voltage on detector circuit. The experiment was conducted with beacon at distances of 1, 5 and 10 feet away from detector.

with its required positive and negative voltages. A single eye was placed in a fixed position and the beacon was placed directly in front of the eye and moved to distances of 1', 5' and 10'. The output of the detection system was measured at each of these distances while the supply voltage was decreased in increments of 0.5 volts. Figure 5-6 shows the result of the experiment for distances of 5 ft. and 10 ft.

As can be seen from the graphs, the decrease in output due to the reduction in supply voltage has a rather linear profile at each of the ranges where measurements were taken. When the supply voltage was decreased below 3 volts, no signal could be detected. In this case, there was a voltage present at the output stage of the circuit, however when the eye was completely covered the voltage did not change, indicating that the circuit was no longer capable of detecting a signal. For the simulation, a linear approximation to these curves was used.

In summary, two experiments have been conducted. The first experiment shows that the measured A/D readings as a function of range hold a $1/r^2$ dependency. The second experiment measured the effects of the processor's input voltages on eye-A/D readings. From this experiment it was conclusively shown that the input voltage has a linear effect on the A/D readings at a given range. The slope of the linear region of the supply voltage vs. normalized circuit output data having a value of 0.0391 $volts^{-1}$ and shown in Figure 5-6, is used to compute the change in processor output as a result of battery voltage degradation. The slope of the linear region for a range of 10 ft. was used because at this range the processor output yielded the highest values at ranges both greater and smaller than 10 ft. source distance.

F. EXTERNAL FACTORS

The detector behavior previously discussed is largely attributed to the internal parameters and characteristics of the processor. However, the one external factor that was discussed is the intensity received by the detector as a function of range. From Figure 5-4 it can be seen that once the maximum saturation distance is reached, the received beacon intensity, as a function of range only, behaves as $1/r^2$. As a result of this distribution, the equation relating the transmitted intensity to the received intensity is given by

$$I_R = I_T * T_A , \qquad (5-2)$$

where T_A is the transmission factor, a fractional value based on the transmission medium. Table 5-1 gives the values of T_A. In the table, r is the range of the beacon from the

CONDITION	VALUE OF T_A
$r < r_s$	1
$r_s \leq r < r_{max}$	$(r/r_s)^2$
$r > r_{max}$	0

Table 5-1. Table showing values of T_A for corresponding range conditions.

detector, r_s is the maximum distance at which the "eyes" are saturated, and r_{max} is the maximum range at which the detector can detect the beacon light. Each of these

variables is an input parameter. The method for measuring each variable is described in Chapter VII.

The positioning and resulting geometry of photodiode positioning account for additional external effects, which play a large role in determining the intensity integer produced by the A/D converter. As previously stated and depicted in Figure 5-1, the robot "eyes" are usually positioned adjacent to each other and are separated by a nose of some length to determine which side the beacon light is being detected from. In addition

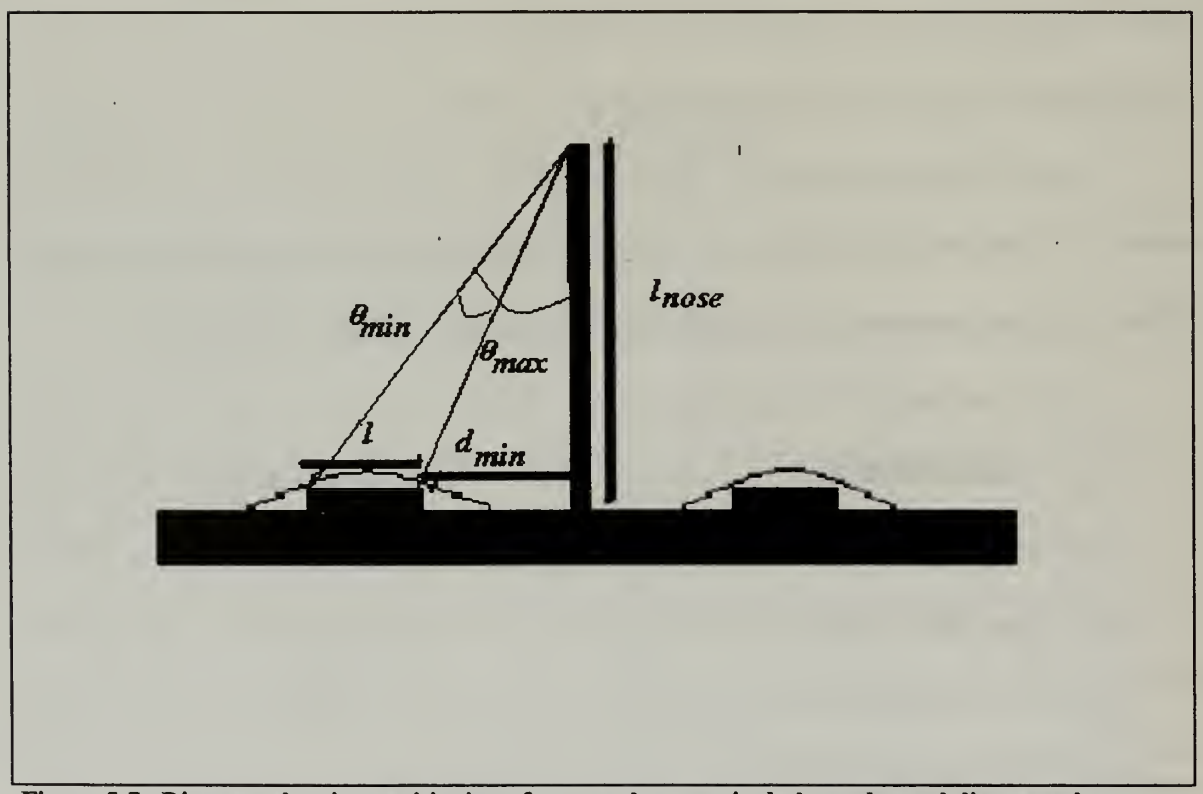

Figure 5-7. Diagram showing positioning of eyes and nose to include angles and distances that account for shadowing and discrimination.

to the expected single photodiode angular dependence on the incident angle of the beacon light, the ratio of nose length to eye distance from nose-base ratio presents another set of

angular dependencies and subsequent effects. Figure 5-7 shows the set of angles introduced by the addition and use of a nose.

For the purpose of the simulation, it has been assumed that the set of eyes is a single point on the robot so that one incident angle is used for both eyes. Using this assumption, if no nose were used, each eye would receive the same amount of light and consequently there would be no way to determine which side relative to the robot the source is positioned. As a result, no decipherable information could be used for robot control.

The angle of incidence, θ, is the angle at which the light from the beacon falls on the active portion of the photodiode. The intensity as a result of reduction in effective area subtended is given by

$$I_\theta = I_R * \cos\theta \ . \qquad (5-3)$$

In this equation, I_R is the light intensity received from the beacon and θ is the incident angle.

With the addition of a nose, if the source were positioned on either side of the nose centerline, one eye receives the total amount of incident light while the other is shielded from the light to some degree. This relationship is illustrated in Figure 5-7. As can be seen from the figure, the extreme ends of the photodiode active area limit the amount of light received by an eye. An eye positioned on the same side as the source receives the full amount of incident light, depending on its angle of incidence. The other eye receives the same amount of light as the first only if the angle of incident light is less than θ_{min}, where θ_{min} is given by

$$\theta_{min} = tan^{-1}(\frac{d_{min}}{l_{nose}}), \qquad (5-4)$$

where d_{min} is the measured distance between the inner extreme of the photodiode active area and the base of the nose and l_{nose} is the measured length of the nose. If the angle of incident light is greater than θ_{max}, the eye receives none of the incident light. The angle θ_{max} is given by

$$\theta_{max} = tan^{-1}(\frac{d_{max}}{l_{nose}}), \qquad (5-5)$$

where d_{max} is the measured distance between the outer extreme of the photodiode active area and the nose base and is given by

$$d_{max} = d_{min} + l. \qquad (5-6)$$

In Equation 5-6, l is the total length of the photodiode active area. The derived formula that gives the fraction of intensity received is given by

$$I_s = I_\theta * [(1 + \frac{d_{min}}{l}) - \frac{L}{l}] \qquad for\ \theta_{min} < \theta < \theta_{max}, \qquad (5-7)$$

where I_s is the intensity received as a result of shading and I_θ is the intensity received as only a result of its angular dependence. If the angle of incident light falls between the two angles, then the intensity of incident light is shaded to some degree based on the amount of photodiode active area actually being shielded from the incident light by the nose. The L in Equation (5-7) is given by

$$L = l_{nose} * tan\theta. \qquad (5-8)$$

Thus, Equation 5-7 gives how much of the photodiode active area is actually being shaded.

G. CONCLUSIONS AND RESULTING TRANSFER FUNCTIONS

From the measurements made on the factors affecting the detector, a series of signal input-to-output transfer functions may now be defined. It has been concluded that, for the purposes of the simulation, the intensity received by either of the "eyes" is primarily a function of three internal entities. The first two factors are the detector intensity's dependence on both range and incident angle, as discussed in the previous section.

The third factor is the detector's dependency on the voltage supplied to the processor, where the signal input to output function is given by

$$I_D = I_R * F_{PI}(V_D), \qquad (5-9)$$

where I_D is the intensity received by the detector, which is dependent on the supply voltage and I_R is the intensity that would be received if there were no voltage effects. F_{PI} is the relating transfer function and is given by

$$F_{PI} = 1.0 - \Delta P_{out}, \qquad (5-10)$$

where ΔP_{out} is the change in processor output and is calculated by multiplying the change in detector supply voltage by the slope of the linear region represented by the data in which the source was 10 ft. from the detector, as shown in Figure 5-6. The quantity

yielded by the transfer function tells the percentage of I_R remaining intact given the reduction in detector supply voltage. The line slope yields a value of 0.0391 volts^{-1}.

The detector battery supply is affected in the same manner as the beacon oscillator battery supply. The matrix shown in Table 4-1 is used to select the battery type and loading condition as before. The simulation defaults are 9 volts for battery type and light loading. As a result, the detector battery supply, V_D, is given by

$$V_D = V_D(t=0) - (5.56 \times 10^{-6} \text{ volts/sec} * \Delta t), \qquad (5\text{-}11)$$

where Δt is the elapsed scenario or detection system operation time.

The fourth critical factor affecting the detector intensity is the frequency of the flashing beacon light. As previously stated, the beacon is tuned to an initial frequency that decreases with time. The detector's circuit is tuned to the initial frequency. The frequency response curve shown in Figure 5-3 shows how the circuit responds to various input frequencies. The equation giving a signal input to output relationship is given by

$$I_f = I_{PI} * T_f(f_0, f_P). \qquad (5\text{-}12)$$

In this equation, T_f is the Gaussian-like transfer function that gives the desired frequency response of the circuit and is given by

$$T_f = \exp\left(\left(-\frac{|\Delta f^2|}{2\sigma^2}\right)\right), \qquad (5\text{-}13)$$

where Δf is the difference between the difference between the received beacon flash frequency, f_0, and processor's tuned frequency, f_P; and σ is the characteristic width of the frequency response curve. The characteristic width is set so that when $\Delta f=0$, the received

intensity is maximum; and when Δf *is* ½ of the desired bandwidth, the intensity is reduced by a factor of 0.707. Thus, σ is defined by

$$\sigma = \frac{f_{tuned}}{2 * Q}, \qquad (5\text{-}14)$$

where f_{tuned} is the frequency to which the processor is tuned.

The last critical factor affecting the received detector intensity is the discrimination and shadowing effect discussed earlier. This factor is also important because it allows for two different values, one for each "eye", to be yielded from a single given input intensity. The function relating the input intensity received by the set of "eyes" to the output intensity received by each eye is given by

$$I_{eye} = I_D (T_{shadow} (\theta, l, d_{min}, l_{nose})). \qquad (5\text{-}15)$$

In this equation, T_{shadow} is the transfer function that uses equation (5-5) to determine shadowing effects for the "eye" that is not on the same side as the source.

H. SUMMARY

This chapter discussed the detector component of the optical detection system. It described the detector's physical characteristics as well as its actual and measured behavior. Transfer functions were defined that allow the pertinent signals received and processed by the detector to be effectively modeled. As a result of these transfer functions, it has been concluded that for an effective simulation there are only five internal and external detector factors that must be considered. These factors are: (1) the angular dependence of the incident light, (2) the range of the beacon from the detector,

(3) the effects of voltage on the processor, (4) the circuit's frequency response and (5) the effects of shadowing and eye discrimination.

VI. TRANSFER FUNCTION SYNOPSIS

A. INTRODUCTION

This chapter organizes and redefines each of the transfer functions defined in the previous two chapters so that they might effectively simulate the step-by-step process of an SE 3015 optical detection sequence.

B. OVERVIEW

As previously stated, the purpose of this thesis is to accurately replicate the optical detection sequence of the robots used in the SE 3015 Robot Wars competition in a computer simulated model. In accomplishing this, the various components of the robots have been described along with defined signal input to output transfer functions for each of the pertinent components. The use of these transfer functions allows one to produce a signal output, in terms of an A/D integer number, that represents a dependency on each of the important internal and external factors of both the beacon and detector.

In order to use the transfer functions, a method of applying each of transfer functions must also be developed. The robot command, control, and detection algorithms utilize the A/D integers produced from each eye, interpreting a range and angle dependence (r,θ) to the opponent's robot. As a result, the simulation assumes the range and relative angle from the combatant Simbot to the referenced Simbot are known at each time step during the detection sequence. Thus, a method of making corrections to the final A/D integer is used rather than operating on internal system signals with functions. The final A/D integer is used by the simulation for Simbot operation and display.

C. TRANSFER FUNCTION FLOW

Since it is the final A/D integer that the simulation uses, the user is asked to input the value of the maximum A/D integer that the robot detector is capable of producing. It has been assumed that both detectors have been calibrated so that the integer reading is the same for both eyes. Procedures for obtaining the maximum A/D integer and all other required input parameters are discussed in the next chapter. The remainder of this chapter discusses, step-by-step, how each of the equations and transfer functions are used in the simulation to produce the ultimate A/D integer reading for each eye.

As occurring on the actual robot, the detection sequence is initiated with an emission of light from the beacon at some intensity and frequency. For this beacon emission, two transfer functions have been defined; one for the beacon's intensity and another for its frequency. The beacon intensity, I_B, is characterized by

$$I_B = I_{max} * F_I(V_B), \qquad (6-1)$$

where I_{max} is the maximum beacon intensity possible under conditions of maximum supply voltage. I_B depends on V_B, the beacon supply voltage and F_I is the related transfer function. This transfer function uses the slope of the linear portion of a curve like the one shown in Figure 4-7 to calculate the change in supply voltage with elapsed beacon operation time. The slope of the linear portion of the curve is given by the default light loading condition and 9-volt battery case as

$$\frac{\Delta V_B}{\Delta t} = 5.56 \times 10^{-6} \text{ volts/sec}, \qquad (6-2)$$

where ΔV_B is the change in beacon oscillator battery voltage and Δt is the elapsed scenario or beacon-operation time. Slopes for other batteries and loading conditions are given in Table 4-1. The data in Figure 4-7 was used to define the transfer function, F_I, which gives the factor by which the maximum normalized intensity of the beacon has decreased as a result of a decrease in supply voltage. The transfer function is given by

$$F_I = 1.0 - \Delta I, \qquad (6\text{-}3)$$

where ΔI is calculated by multiplying the change in beacon supply voltage by one of the slopes given by the linear regions of Figure 4-7. Two of the three different regions used have been outlined on the figure. The slopes and their respective voltage regions are given in Table 4-1.

In addition to the simulated beacon intensity, the simulated beacon flash frequency must also be transmitted to the detector. The simulated frequency is given by

$$f_B = f_o - F_f(V_B), \qquad (6\text{-}4)$$

where f_o is the frequency at which the beacon oscillator is initially tuned at the beginning of a detection sequence. The term f_B is the transmitted beacon frequency, which is dependent on the beacon supply voltage and F_f is the related transfer function. The transfer function is given by

$$F_f = \Delta V_B \,/\, 23.15 \text{ Hz per volt}, \qquad (6\text{-}5)$$

where the value 23.15 Hz per volt is the slope of the linearly approximated data in Figure 4-6.

Now that the simulated beacon's intensity and frequency have been determined, the intensity must be corrected for decrease with distance using T_A, the transmission factor for air. This is given by

$$I_T = T_A * I_B , \qquad (6\text{-}6)$$

where T_A has the values defined in Table 5-1.

At this point, the intensity has been transmitted by the beacon. As a result of the intensity being received by the detector, it is now transformed into an A/D integer number that represents the intensity received by the set of detector eyes. Prior to running the simulation, the user is required to input the maximum A/D integer, AD_{max}, that can be produced by the actual A/D converter as a result of light received from a beacon at full intensity, with no dependence on V_B. The default value for AD_{max} is 4080. The transformation is made by

$$AD_0 = AD_{max} * I_B, \qquad (6\text{-}7)$$

where AD_0 represents the highest possible A/D integer than can be produced by the detector as a result of light received from a beacon whose intensity is dependent on V_B.

As discussed in Chapter IV, there is a background or noise intensity that is received by the detector in addition to the intensity of the beacon itself. This intensity is applied in terms of A/D integer, so that the full A/D intensity received by the detector is

$$AD_{total} = AD_0 + AD_N, \qquad (6\text{-}8)$$

where AD_N is the minimum A/D integer produced by the detector. This value is a user input parameter, for which the method of measurement is described in the next chapter.

The angular dependence is then applied to the total A/D number. The equation producing this dependency is given by

$$AD_1 = AD_{total} * T_\theta, \qquad (6\text{-}9)$$

where T_θ is given by

$$T_\theta = \cos\theta, \qquad (6\text{-}10)$$

where θ is the angle at which the beacon light is subtended on the active area of the photodiode. This is the lone factor on the beacon light subtended, due to the negligence of a lens effect as discussed in the previous chapter.

At this point, two of the three external factors affecting the received detector intensity have been applied. The internal factors are now applied to the A/D integer, AD_1. The detector battery supply, V_D, is affected in the same manner as the beacon oscillator battery supply. The matrix shown in Table 4-1 is used to select the battery type and loading condition as before. The simulation defaults are 9 volts for battery type and light loading. As a result, the detector battery supply is given by

$$V_D = V_D(t=0) - (5.56 \times 10^{-6}\ volts/sec * \Delta t), \qquad (6\text{-}11)$$

where Δt is the elapsed scenario or detection system operation time.

Since one of the two internal factors is dependent on the detector supply voltage, and the defaults used are for a 9-volt battery under light loading conditions, the slope given Equation 6-2 is used also to compute the degraded voltage. The equation that computes the detector's dependency on voltage supplied to the processor is given by

$$AD_2 = AD_1 * F_{PI}(V_D), \qquad (6\text{-}12)$$

where AD_2 is the yielded intensity, which is now dependent on the supply voltage. AD_2 is the previous A/D intensity from Equation 6-11 that would be received if there were no voltage effects and F_{PI} is the relating transfer function. F_{PI} is given by

$$F_{PI} = 1.0 - \Delta P_{out}, \qquad (6\text{-}13)$$

where ΔP_{out} is the change in processor output and is calculated by multiplying the change in detector supply voltage by the slope of the linear region represented by the data in which the source was 10 ft. from the detector, as shown in Figure 5-6. The quantity yielded by the transfer function tells the percentage of I_R remaining intact given the reduction in detector supply voltage. The line slope is given by

$$\frac{\Delta P_{out}}{\Delta V_D} = 0.0391\ volts^{-1}. \qquad (6-14)$$

The final Simbot internal detector factor, the processor's frequency response, must now be applied to the previously yielded A/D integer. The frequency response curve shown in Figure 5-3 shows how the circuit responds to various input frequencies. The equation giving a signal input to output relationship is given by

$$AD_3 = AD_2 * T_f(f_B, f_P). \qquad (6-15)$$

In this equation, T_f is the Gaussian-like transfer function that gives the desired frequency response of the circuit and is given by

$$T_f = exp((-\frac{|\Delta f^2|}{2\sigma^2})), \qquad (6-16)$$

where Δf is given by

$$\Delta f = f_B - f_P. \qquad (6-17)$$

In Equation 6-16 and 6-17, σ is the characteristic width of the frequency response curve and f_P is the processor's tuned frequency. The characteristic width, σ, is set so that when $\Delta f=0$, the received intensity is maximum and when Δf *is ½* of the desired bandwidth, the intensity is reduced by a factor of 0.707. Thus, σ is defined by

$$\sigma = \frac{f_{tuned}}{2 * Q}, \qquad (6\text{-}18)$$

where f_{tuned} is the frequency to which the processor is tuned.

The last step in the process of producing the ultimate A/D intensity readings for each Simbot eye is to apply the effects of nose shadowing and discrimination to the previous yielded A/D reading. The function relating the input intensity received by the eyes to the output intensity received by each eye is given by

$$AD_{eye} = AD_3 * T_{shadow}(\theta, l, d_{min}, l_{nose}) \qquad (6\text{-}19)$$

The variables l, d_{min}, and l_{nose} are illustrated in Figure 5-7 and discussed in detail in Chapter V. T_{shadow} produces the ultimate A/D intensity reading for each Simbot eye. The details of the computations involved in this transfer function are also discussed in Chapter V.

D. SUMMARY

In summary, each of the transfer functions used to generate the ultimate A/D intensity readings for each Simbot eye has been discussed again. The simulated beacon intensity and frequency are the transmission component factors represented by transfer functions. The Simbot detector has both internal and external factors that are represented by equations and transfer functions. These factors are the intensity's range and angular dependence, the effects of processor supply voltage on the intensity, the processor's frequency response and the effects of shadowing and eye discrimination on the intensity.

VII. SOFTWARE IMPLEMENTATION

A. INTRODUCTION

This chapter discusses how each of the previously defined transfer functions are implemented into the source code for effective simulation of a series of Simbot optical and tracking detection sequences.

B. CODE DEVELOPMENT

Keeping in mind the process for utilizing the final A/D integer discussed in the previous chapter, the maximum A/D intensity can be processed by each of the transfer functions previously discussed to compute the final A/D reading for each Simbot "eye". In addition to the maximum A/D intensity, other parameters must be known in order to use the transfer functions. To obtain these parameters, the user of the model is asked to input a number of values into the program. Table 7-1 shows a list of all the robot and beacon parameters that the user is asked to provide. Because each the parameters is member of the Simbot class, the user must input the required values prior to starting the simulation.

In previous work done for the simulation, a program was written in the Java programming language that integrated the equations of motion for robot movement and gave robot positional reports based on commands issued to the Simbot. The same program was modified so that it can be used for the detection sequence simulation. In the simulation, the time steps and subsequent evolution of time through the scenario

corresponds to real seconds, so that 5 seconds of Simbot movement is the same as 5 seconds of actual robot movement. At each of the integration time steps, the range and relative angle of beacon to Simbot is computed, along with the total time of Simbot operation. Given this information, the remainder of this chapter goes on to describe how the transfer functions are implemented in the simulation. The Java code used for the simulation is listed in Appendix A.

In the program, the function ***compute_pos*** integrates the equations of motion to provide Simbot positional data. Within the function itself, at the end of each positional computation, the process of "sampling" the input to the detector to provide left and right "eye" A/D readings takes place. Before the actual "sampling" process can take place, the range and the relative angle of the beacon to the Simbot is computed. The function ***beac_rel_angle*** takes the position of the robot and the position of the beacon and computes relative angle, while the function ***beac_range*** takes the same positions and computes the range.

The sampling process is initiated by computing the beacon transfer function, which consists of the beacon frequency and the fraction of total beacon intensity. The function ***bcn_xmit_int*** uses the measured initial voltage of the beacon oscillator and the time elapsed during the scenario to compute the fraction of total intensity that the beacon in capable of transmitting. The function compares the simulated beacon supply voltage to the appropriate region and applies the corresponding slope as discussed in Chapter VI. Equations 6-3 and 6-4 are used to compute the intensity transmitted by the simulated beacon.

The beacon frequency as a function of elapsed time is calculated similarly. The function ***bcn_flsh_freq*** takes the measured, initial beacon supply voltage; the elapsed

PARAMETER NAME	*DEFINITION*	*METHOD OF MEASUREMENT*	*DEFAULT VALUE*
MAX_AD_INT	Integer value of maximum A/D intensity, detector is capable of receiving	Place beacon directly in front of eye at closest possible position record value	4080
MIN_AD_INT	Integer value of minimum A/D intensity read when there is 100% probability of detection	Place beacon and eye at same height, maximize distance between the two until integer just above noise integer is read, record value	48
MAX_NOISE_INT	Integer value of maximum A/D reading present when there is no beacon present	Observe A/D readings when beacon is off, record maximum value	32
MIN_NOISE_INT	Integer value of minimum A/D reading present when no beacon is present	Observe A/D readings when beacon is off, record minimum value	16
MAX_SAT_DIST	Float value of maximum distance where MAX_AD_INT can still be read	Place beacon and eye at same height, increase distance between the two until integer reading falls just below MAX_AD_INT, record distance in feet	5.0 ft
MIN_DET_DIST	Float value of maximum distance where beacon can still be detected, 100% prob. of detection	Perform measurement for MIN_AD_INT, record distance in ft.	20.0 ft
NOSE_LENGTH	Float value of height or length of nose	Measure nose length in inches	1.5 inches
TOT_EYE_SEP	Float value of total distance between active area of the two eyes	Measure total distance between inner extremes of active eye area in in.	1 inch
INIT_BCN_VOLT	Float value of initial battery voltage for beacon oscillator	Measure beacon oscillator battery voltage	8.5 volts
INIT_BCN_VOLT	Float value of initial battery voltage for electrical circuit	Measure electrical circuit battery voltage	8.5 volts

Table 7-1. Table of user Simbot input parameters. These parameters allow use of transfer functions in the simulation. If no value is provided, each parameter has its own default value.

scenario time; and the tuned beacon oscillator frequency and computes the degraded frequency of the beacon by using Equations 6-4 and 6-5.

The next functions in the simulation compute the detector fractional values for the degraded processor's supply voltage and the circuit's frequency response factors. For the detector circuit supply voltage factor, the function ***det_sply_volt*** is used. Equation 6-11 is used to calculate the degraded processor voltage.

The function ***filter*** takes the tuned electrical circuit frequency, the beacon frequency, and σ, the characteristic width of the curve and computes the fraction of total intensity the detector is capable of outputting as a result of the difference between the two frequencies. In effect, it determines the frequency response of the circuit. Equation 6-16, which represents the filter transfer function, is used to compute the fractional intensity.

The final function used to calculate the A/D intensity reading for each "eye" is called ***bot_detector***. This function is responsible for determining all of the external parameters affecting the intensity readings. The function itself determines if the beacon falls within view of the detector "eyes" . It uses the functions ***eye_ang_dep*** and ***discriminate*** to determine the angular dependency of the intensity and to discriminate between the left and right eyes, respectively. The latter of these functions also computes the effects of shadowing. The ***bot_detector*** function uses the parameters for range, beacon frequency, angle of incidence, I_{B_n}, F_f, and F_{PI} to compute the intensity reading for each Simbot "eye". As the function is entered, the following variables are declared and initialized.

```
int rcvd_int = MIN_NOISE_INT; //gives default value set of eyes
int LEFT_EYE_INT = rcvd_int; //gives default value to left eye
int RIGHT_EYE_INT = rcvd_int; //gives default value to right eye
boolean reye_detect = false; // right eye, no initial detection
boolean leye_detect = false; // left eye, no initial detection
```

Because each eye has a 90° field of view (FOV), the angle of incidence is compared to the field of view limits for each eye. If the incident angle falls within the FOV of either eye, the boolean condition value initialized above is set to true. When the conditional value is set to true for either eye, a detection may have occurred and as a result, the following series of tests is initiated.

First of all, if the range is greater than maximum detection distance, the conditional value is set back to false, the series of steps is immediately ended, and both the eye values remain as initialized. If that condition is not met, then the range is compared to the maximum saturation distance range. If the range is less than or equal to ***MAX_SAT_DIST***, then the value of ***rcvd_int*** is assigned as

$$rcvd\_int = I_B * F_V * F_f * MAX\_AD\_INT \qquad (7\text{-}1).$$

However, if the range is greater than ***MAX_SAT_DIST***, then the value of ***rcvd_int*** is assigned as

$$rcvd\_int = I_B * F_V * F_f * T_A * MAX\_AD\_INT \qquad (7\text{-}2).$$

Once the value of ***rcvd_int*** is calculated, the function ***eye_ang_dep*** is then used to determine and the angular dependence of the detector received intensity by using Equation 6-10. The value returned by the function is the angular dependent intensity received by the set of detector "eyes".

In order to compute the intensity readings for each "eye" a function must be used to discriminate the intensity values received by the left and right "eyes". The function ***discriminate*** performs the necessary calculations, by taking the value of the intensity

received by the "eye" set, the angle of incidence, and the boolean value of either ***leye_detect*** or ***reye_detect***. The function is called twice and subsequently returns two different values. This first call is for the left eye, returning the integer value of the left eye's A/D reading, while the second call does the same for the right eye. The ***discriminate*** function computes the values for θ_{min} and θ_{max} defined by Equations 5-4 and 5-5, respectively. The boolean value for either ***leye_detect*** or ***reye_detect*** is then tested. If the value is true, the function returns and that particular eye gets the same value as the intensity received by the set of "eyes". If the value is false, the following three conditions are tested.

In the first condition, if the incident angle is less than θ_{min} the eye's A/D reading is assigned the same value as the intensity received by the set of "eyes". In the second condition, if the angle of incidence is greater than θ_{max}, the eye's A/D reading is assigned ***MAX_NOISE_INT***. For the last condition, if the angle of incidence lies between θ_{min} and θ_{max}, the eye is shadowed to some degree and is thus assigned the value calculated by Equation 6-19, which gives the intensity as a result of nose shadowing. The function ends with the computed A/D intensity of the eye being returned. As previously stated, the function is called twice to return values for both the left and right eyes.

C. SUMMARY

In summary, the method of actually implementing the previously developed transfer functions into the simulation code has been discussed. For this implementation, an earlier Java program that computed the Simbot's position based on its equations of motion was modified so that at each integration time step, the light emitted from the

simulated beacon is "sampled" by the detector, and the resulting A/D intensity reading from the Simbot's "eyes" is produced. The steps in between the initial beacon emission and the final A/D reading are performed through use of the transfer functions derived in the previous sections. In the code, each of the transfer functions is represented by a Java function that passes the parameters necessary for computing the desired transfer function output, which is returned. Table 7-2 shows each of the functions used and its corresponding transfer function. The successful generation of A/D intensity reading for each eye, now allows for the data to be used in algorithms for Simbot command, control, detection, and tracking.

FUNCTION NAME	*TRANSFER FUNCTION REPRESENTED*	*ARGUMENTS PASSED*	*OUTPUT*
bcn_xmit_int	I_B	Initial beacon battery voltage & elapsed time in scenario	Fraction of total intensity detector is capable of receiving as a result of decrease in beacon intensity
bcn_flsh_freq	F_f	Initial beacon battery voltage, elapsed time in scenario & tuned beacon frequency	Degraded beacon frequency as a result of time
det_sply_volt	V_D	Initial beacon battery voltage & elapsed time in scenario	Fraction of intensity detector is capable of receiving as a result of its degradation is supply voltage
filter	T_f	Tuned detector frequency, current beacon freq. & characteristic width of frequency response curve	Fraction of intensity detector is capable of receiving as a result of its frequency response
bot_detector	none	Detector range from beacon, current beacon frequency, T_f, T_B, & F_V	A/D intensity reading for each eye
eye_ang_dep	T_θ	On-axis received intensity for set of eyes	Angular dependent intensity for eye set
discriminate	T_{shadow}	Angle of incidence, off-axis intensity for eye set & true/false condition confirming detection for individual eye	Received intensity for individual eye as a result of shadowing after discrimination between eyes

Table 7-2. Table showing each of the Java functions used to represent the derived signal input to output transfer functions. The combined set of these functions allows for generation of A/D intensity readings for each eye.

VIII. SIMULATION PERFORMANCE VERFICATION

A. INTRODUCTION

This chapter describes how each of the transfer functions implemented for the effective simulation of the SE 3015 robot optical detection system and sequence were verified for accurate behavior and performance.

B. PERFORMANCE TESTS

As previously stated, the purpose of this thesis is to develop a simulation that accurately and effectively replicates the behavior of the present optical detection system used on robots for the SE 3015 Robot Wars Competition. In order to analyze performance, a series of tests was conducted on each of the program functions listed in Table 7-2. To conduct these tests, two scenarios were developed for the Simbot to operate in. In the first test, the Simbot was given a set of commands from a scripted scenario and a simulated beacon was placed at various stationary distances away from the Simbot's initial position. The total time of the scenario was 39 seconds. In the second, the Simbot and simulated beacon were placed in fixed positions so that the distance between the Simbot and beacon was greater than the maximum saturation distance and less than the maximum detection distance. The values of the robot input parameters listed in Table 6-1 were varied and the resulting Simbot generated A/D data analyzed and compared to expected results. The tests conducted may also be compared to some of the

experiments conducted in determining the pertinent features of the detector and transmitter needed for effective modeling.

First of all the ***bcn_xmit_int*** function was tested for effectiveness. This test entailed simulating the "the turning on and off of the beacon" using the scripted scenario and a check to ensure that the intensity of the beacon was decreasing with time due to battery drainage using the fixed position scenario. The first test was accomplished by simply changing the value of the input parameter for the initial beacon battery voltage, **BCN_SPLY_VOLT.** When the value was initially at its nominal value of 8.5 volts, the Simbot behaved normally. As expected, the data revealed no noticeable decrease in intensity over this short operation period. When the input value was decreased to its minimum of 1.5 volts, the simulated beacon behaved as if it was "off". As the Simbot moved through the scenario, it's A/D readings remained at the minimum noise level of 16 as expected. The degradation of the simulated beacon's intensity with time was checked by simply observing the value of the beacon's normalized intensity, at each time step, as the scenario progressed. As expected, the value decreased with progressing scenario time.

The functions ***bcn_flsh_freq*** and ***filter*** were tested simultaneously using the fixed position scenario because each one's unique dependency on the other. When the beacon is tuned to any frequency other than the frequency to which the electrical circuit is tuned, there is expected to be a decrease in the frequency response of the detector filter. This decrease in frequency response results in a decrease in the Simbot's A/D readings. To test these expected results the input parameter for the tuned beacon frequency was varied to two different values. The first value was the exact same frequency as that to which the

detector was tuned. As expected, the Simbot's A/D readings corresponded to their maximum possible values given other dependencies such as source range and incident angle. The second value used was the lower bandwidth of the frequency response curve, f_l, given the assigned beacon frequency and a Q of 50. As expected, when the same test was conducted as before the, Simbot's A/D readings were initially reduced from their maximum values by a factor of 0.707. The simulated beacon's frequency degradation with voltage and time was measured by simply observing that the frequency did indeed decrease with progressing scenario time. The result of this decrease in simulated beacon frequency also affected the frequency response of the detector and the resulting Simbot's A/D readings.

The program function ***det_sply_volt*** was tested by simply observing the decrease in Simbot detector supply voltage with progressing scenario time using the fixed position scenario. Because the actual A/D readings only decreased slightly due to the relatively short scenario time, the actual value of the transfer function represented by ***det_sply_volt***, V_D, was observed to ensure that it decreased as expected. The results of the test were conclusive and agreed with the expected results.

The ***bot_detector*** function contains the ***eye_ang_dep*** and the ***discriminate*** functions. The function itself uses the distance between the Simbot and beacon and the angle at which the beacon light is incident on the Simbot "eyes" to determine whether the Simbot will be capable of generating A/D data. If the determination is made that A/D data can be generated by the Simbot's detector, the ***eye_ang_dep*** and ***discriminate*** functions are used to determine the intensity's angular dependence and the effects of nose shadowing, respectively. These functions were tested using the scripted scenario. The

simulated beacon was placed at a position representing the middle of the Robot Wars Competition playing field and the Simbot positioned at one of the playing field corners. Figure 8-1 shows the positional setup used to test the ***bot_detector*** function. The script gave the Simbot commands, which positioned it at various relative angles and distances from the simulated beacon. The observed data consisted of the range of the Simbot from the beacon, the angle at which the beacon light was incident on the Simbot detector's "eyes", and the resulting A/D intensity reading for each Simbot "eye". As the Simbot made its approach toward the simulated beacon the left eyes' A/D value remained at the minimum noise level, while the right eyes' A/D value increased as the Simbot got closer to the beacon. As the Simbot passed the beacon, both A/D values fluctuated

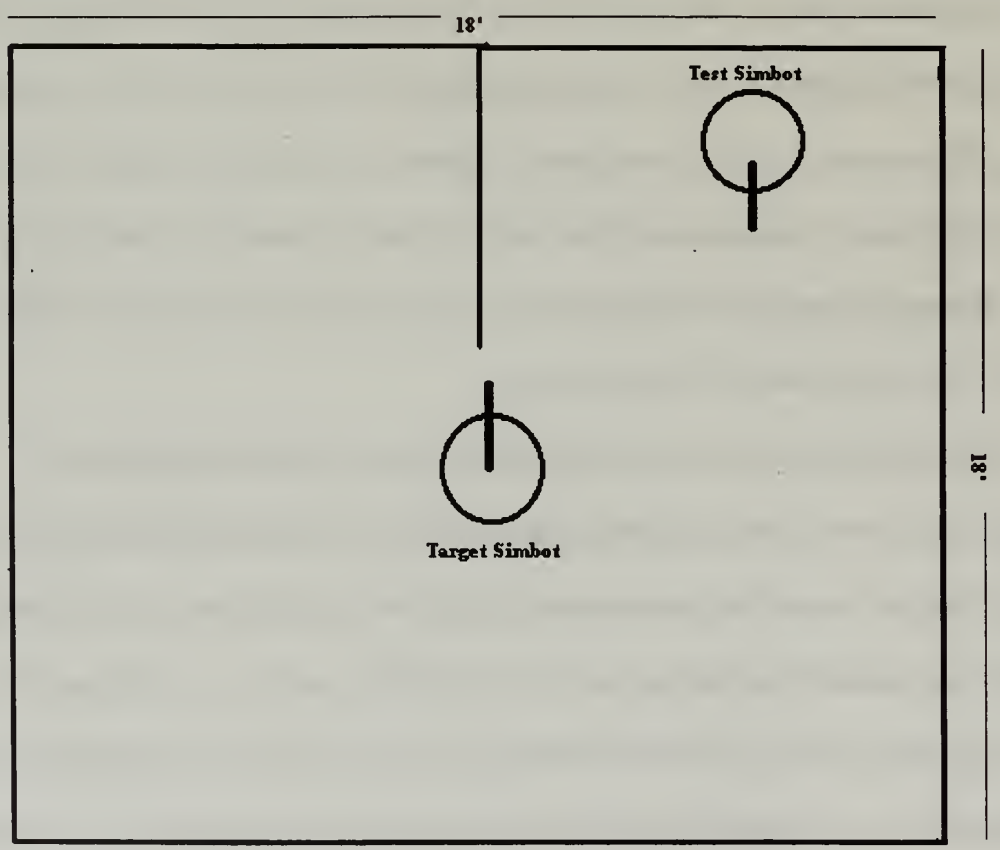

Figure 8-1. Simbot performance test setup.

between the minimum and maximum intensity noise levels. As the Simbot spun on its own axis (both wheels at same angular velocity, in opposite directions), the values of each eyes' A/D reading was observed, to check the effects of the shadowing function. The observation revealed that the value for a completely shadowed eye would slightly increase as that eye came closer to being in a head-on position with the beacon; and that the value for the eye which had been in a head-on position would start to decrease as the Simbot turned away from the beacon. The A/D readings for both "eyes" were observed during the entire Simbot scenario. The entire series of A/D readings did indeed match the positional and rotational locations of the Simbot in each instance.

C. CONCLUSIONS

In order to test the defined transfer functions for effectiveness in simulating robot behavior, each of their corresponding Java code functions were tested. The program functions were tested by using a Simbot in two separate scenarios. Each of the transfer function values and the resulting A/D data from each Simbot "eye" were observed while the Simbot completed its given scenario. In addition to performing these tests, during the development of each program function, the Simbot was given various other simple tests and scenarios, which gave ample indication of accurate Simbot A/D data. From the series of tests conducted on the Simbot and analysis of the resulting A/D data, the conclusion may be drawn that the quantitative simulation is effective in replicating the performance of the actual SE 3015 robot.

D. SUMMARY

This chapter described how each of the defined transfer functions for simulating the SE 3015 robot optical detection system and detection sequence were verified to ensure accurate behavior and performance.

IX. SIMULATION OPERATION

A. INTRODUCTION

This chapter describes how the total SE 3015 Robot simulation operates. In addition the simulation architecture is given along with the functionality of each of the simulation components.

B. OVERVIEW

The purpose of the simulation is to provide students with a tool that allows them to design SE 3015 robot detection parameters and to evaluate the effectiveness of these parameters as well as robot tactics and behavior through use of a graphical model that replicates the interactions that take place between a robot and its target. The simulation manipulates two robot objects (Simbots). The first Simbot is given a program function that simulates the detection algorithm used with an actual robot. The second Simbot is given a scripted scenario by which its movement is based, so as to act as a target for the first Simbot. All of the positional data for each of the Simbots is directed by the simulation to output position interpolator files. The graphics program uses the position interpolator files to recreate each of the Simbot's movements, thus replicating the detection sequence with a 3-dimensional animation display. It should also be noted that it is possible the run the simulation in a mode where neither of the Simbot's is scripted so As to emulate the acutal Robot Wars competition. Figure 9-1 is an illustration of the Simulation architecture. A flow chart of the simulation processes is shown in Figure 9-2.

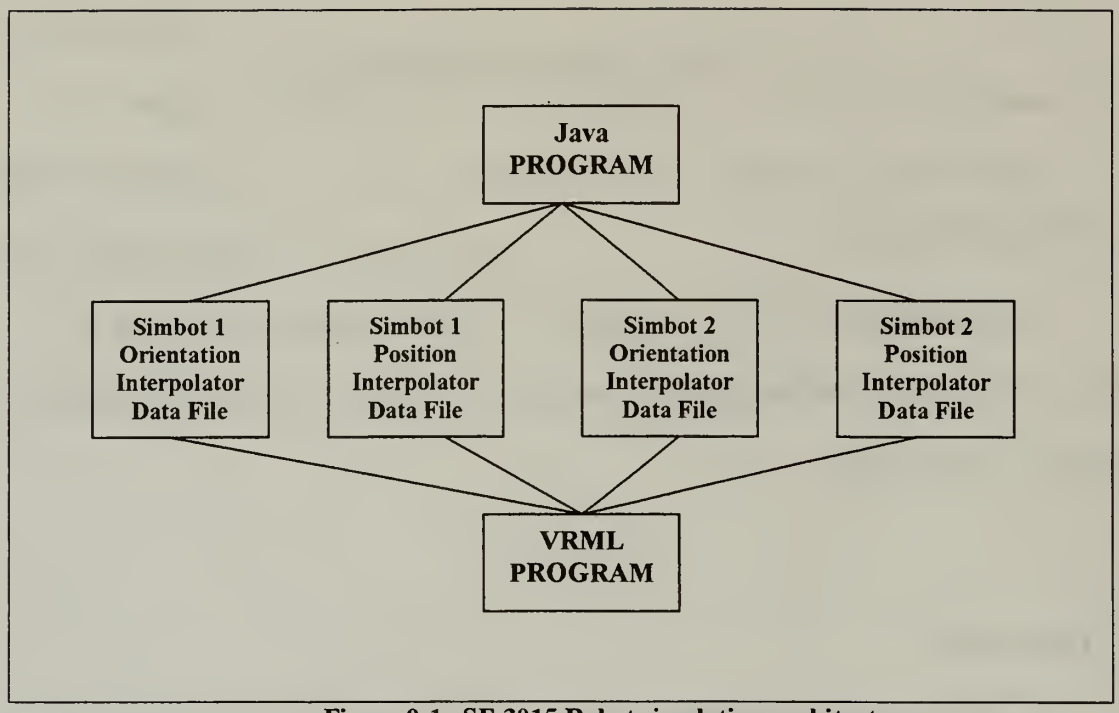

Figure 9-1. SE 3015 Robot simulation architecture.

The total simulation consists of a primary program that computes each of the Simbot's positions, Simbot positional and orientation interpolator files, and a graphics program that generates the animation display. The primary program is written in Java and uses two Simbot objects, as previously discussed. The detection algorithm written in Dynamic C for the SE 3015 Robot Wars was converted into a Java function that replicates its logic and gives the Simbot the same behavior as the actual robot. The Simbot position interpolator files generated by the primary program are ASCII text files. The secondary program is the graphics program used to animate the Simbots, which was written in VRML (Virtual Reality Modeling Language). This VRML program uses the interpolator files to generate an animated replication of the SE 3015 Robot Wars playing-field and a Simbot engaged in a detection and targeting sequence with another Simbot.

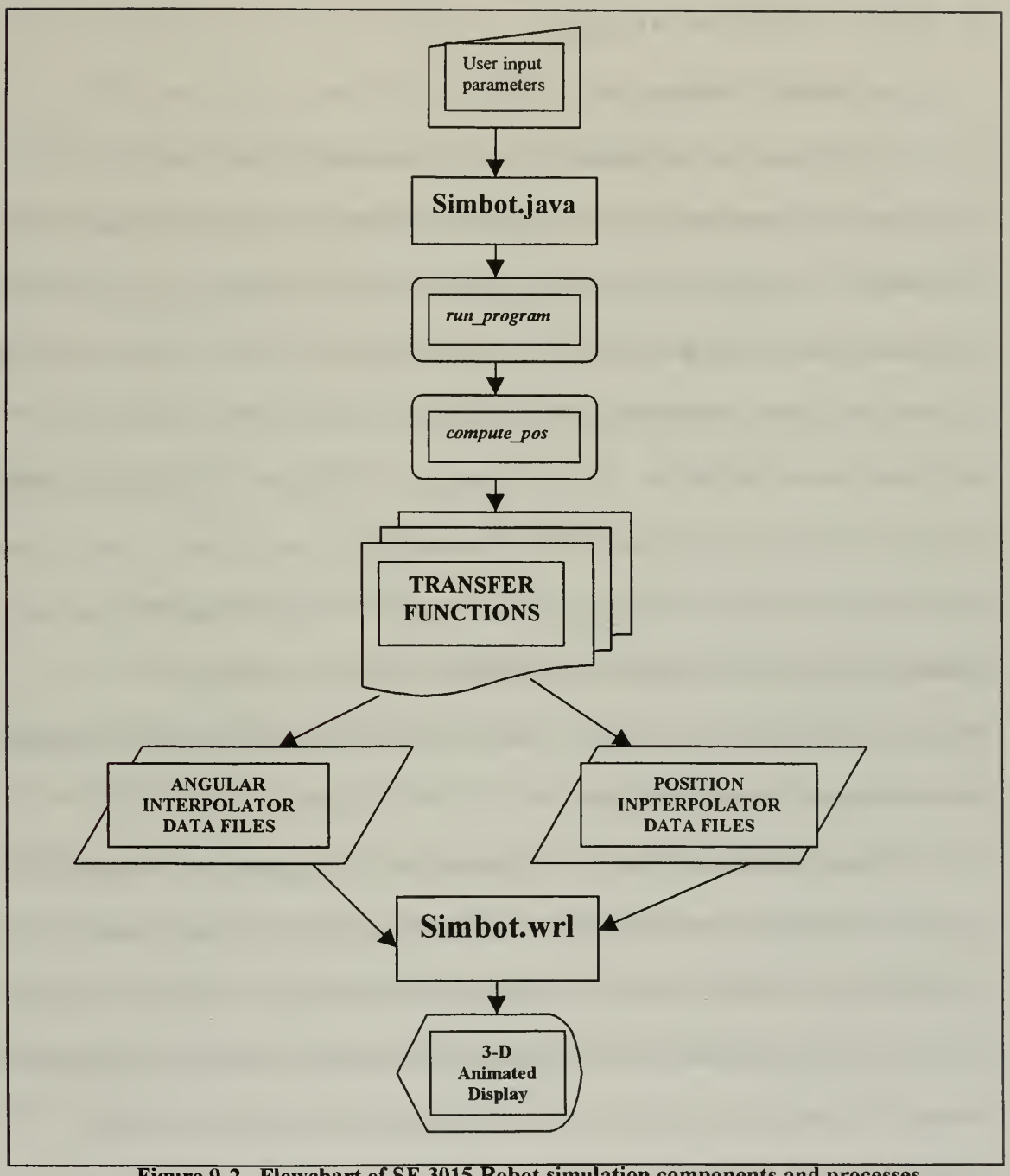

Figure 9-2. Flowchart of SE 3015 Robot simulation components and processes.

The remainder of this chapter discusses each of the programs and interpolator files noted above.

C. SIMULATION COMPONENTS

1. Primary Program

As previously stated, the primary program is written in Java. The Java code instantiates two Simbot objects and subsequently computes the position and heading of each Simbot. A detection and targeting algorithm written in Dynamic C is converted to a Java function called **run_program**. This function gives the first Simbot the same logical behavior as its actual robot counterpart and is used to determine the next movement command given to the Simbot. The modifications to the Dynamic C detection algorithm required for use in the simulation shall be discussed later in this chapter. A script is made out for the second Simbot. This script simply contains Simbot commands that allow the Simbot to behave in some preconceived manner, in terms of position and heading, to be determined by the user. The method for developing a Simbot script shall also be discussed later in this chapter.

For the primary Java program, a previously written program that computes the equations of motion for one Simbot based on commands given in a Simbot script and produces the resulting positional and heading output was modified. The modifications first of all, allow two Simbots to be instantiated and secondly, implement the physics and components needed for an optical detection system and a detection and tracking algorithm for one of the Simbots. During each of the time steps for computing the first Simbot's position, its detector attempts to sample the beacon light from the second Simbot. A resulting A/D intensity reading for each of its eyes is yielded. The readings

for each eye are then used by the **run_program** function to generate the next movement command that will be issued to the Simbot.

This process goes on until the total scenario time is reached. For the scenario time, each simulation and scenario second correlates to an actual second. Time steps for the equations of motion are 0.01 seconds, the Simbot executes each command issued to the Simbot for a duration of 0.1 seconds, and at every 0.5 second interval the positional and heading data for each Simbot is output to positional and angular interpolator data files that are used by the VRML software to generate the animation scene. The Java source code is listed in Appendix A.

2. Graphics Program

The VRML program is used to generate the 3-dimensional (3D) graphics scene of the Simbots and Robot Wars playing-field for the simulation. The program uses primitive VRML nodes to generate a fairly accurate depiction of each of the objects with which the simulation is concerned. The contents of the interpolator data files generated by the Java program are "copied and pasted" into the VRML program and used to animate each Simbot's movement with respect to position and heading on the Simbot playing-field. Samples of the interpolator output files are listed in Appendix B. The VRML source code is listed in Appendix C.

3. Dynamic C Detection Algorithm Conversion and Implementation

The primary program's Java function, **run_program** contains a modified version of the user's, Dynamic C written, robot detection and tracking code. The remainder of

this section discusses how the Dynamic C code is converted to the **run_program** function and how the algorithm is implemented within the primary program.

Each SE 3015 robot is equipped with two, dual-drive servo motors. Each motor is capable of operating at 15 different speeds, in both the forward and reverse modes. The Dynamic C programming language is the language in which the robot communicates. The language's **platform** function accepts robot commands and issues them to the robot for action. The robot command is issued as an eight character motor control string, in the Dynamic C function **strcpy**, that follows the format specified in [Jones, 1997]. An example command string is

"f f g r f f g r".

In this string the first four characters represent left robot wheel commands and the last four characters represent right robot wheel commands. A further breakdown of the motor control string is provided in Table 9-1.

For the simulation, the **platform** and **strcpy** functions are replaced by a **compute_pos** function that takes the Simbot commands and computes the resulting Simbot position. Because the 3rd string byte (motor status) is somewhat a duplication of the 1st byte (wheel speed) and the fourth byte (wheel distance counter) is not used, the **compute_pos** function only takes two values, the 1st and 5th string values in decimal format. The forward or reverse byte is incorporated into these bytes through used of a sign for magnitude, so that a negative sign would indicate a reverse command. As a result of these modifications, the "ffgrffgr" command would be issued to the Simbot in the simulation as

compute_pos(15,15).

Motor Control String

Robot receives a motor control string to control its direction and speed. The string is 8 bytes long and is broken down as listed below. For example, control string "ffgrffgr" issues a command for both motors to go, rotate, forward at speed 15.

ffgrffgr

1st byte left wheel speed (hexadecimal values 0-f corresponding to 0-15)
2nd byte left motor direction (f = forward, r = reverse)
3rd byte left motor status (g = go, s = stop)
4th byte left wheel distance counter (generally not used)
5th byte right wheel speed (hexadecimal values 0-f corresponding to 0-15)
6th byte right motor direction (f = forward, r = reverse)
7th byte right motor status (g = go, s = stop)
8th byte right wheel distance counter (generally not used)

The robot code sends this to an internal motor function by use of a string copy function to a buffer string via the **platform** function. The motor function breaks down the string as listed above and sends the direction and speed information to the left and right motors which will then turn the robot wheels. The functionality can occur any place in the robot program and as often as needed.

Table 9-1. Motor control string format, [Jones, 1997].

In Dynamic C, the **ad_rd8** function samples the A/D intensity reading received by each robot eye and returns the resulting value. As stated previously, in the simulation the simulated target beacon from the second Simbot is sampled at each integration time step. The **run_program** function takes these two integer values as an input and uses the Dynamic C robot detection and tracking algorithm logic to issue the Simbot its next command. While "for loops" are used in the Dynamic C code to extend the amount of time a robot executes a given command, they may also be used within the **run_program** function. The ultimate function of the **run_program** method is to accurately replicate

and implement the exact logic used in the Dynamic C algorithm, using Java code that is much easier to understand and comprehend than its Dynamic C counterpart. Appendix D shows a sample of Dynamic C robot detection and tracking logic. Appendix E shows its Java Simbot counterpart.

4. Simbot Script

The script from which the second Simbot's commands are taken is simply composed of a series of **compute_pos** functions that enable that Simbot to move in a distinct pattern to be predetermined by the user. An example of a simple pattern would be, if the user wanted the Simbot to move out straight for 5 seconds and then come back. This set of movements or pattern is simply defined by outlining the list of commands that allow the robot to execute the pattern. An example Simbot script is shown in Appendix F.

D. SUMMARY

In summary, the actual Simbot simulation and is components have been discussed. The simulation is composed of a primary and secondary program. The primary program is the Java code that instantiates two Simbots and their beacons and computes their positional and rotational status during a detection and tracking sequence. This Java code also implements the physics and components of an optical detection system, to include its Dynamic C detection algorithm, for one of the Simbots. The positional and rotational data computed is output to positional and angular interpolator data files. The secondary program consists of code written in VRML that is "cut and

pasted " into the interpolator data files and used to generate a 3-dimensional animation of a simulated Robot Wars detection and tracking sequence between two Simbots.

X. CONCLUSIONS AND FUTURE WORK

A. CONCLUSION

One of the primary purposes for the SE 3015 Robot Wars simulation project is to evaluate the use and effectiveness of modeling and simulation as an aid to the overall systems engineering process. During the SE 3015 course, over 50% of each student's time is consumed designing and testing the robot's optical detection system. The detailed model of that system, discussed and implemented in this thesis, will provide students with the ability to test and evaluate both perspective and actual robot detection and tracking parameters in a synthetic environment, either alone or against a user controlled Simbot.

The simulation provides its user with fairly accurate graphical depiction of an actual SE 3015 robot and includes the physics and logic necessary for accurate robot movement, detection, and response to an external stimulus. The simulation is capable of integrating the following features: implementation of actual Dynamic C detection and tracking code; realistic optical detection system parameters, performance, and behavior; replicated real-time visualization; accurate robot movements.

A previously written Java program by the author that computes the actual robot motion equations for one Simbot serves as the foundation for this simulation. The program includes measured and calibrated robot wheel angular velocities, wheel diameters, and placeholders for acceleration and deceleration values. That program was modified so that two Simbots are instantiated and "endowed" with optical detection systems for acquisition and detection. User input parameters define the limits and

capabilities of the simulated detection system. A transfer function approach is used to replicate the series of processes between the initial beacon light emission and the final conversion of binocular reception of that signal to A/D integer numbers representing the light's received intensities. The final A/D numbers generated by each Simbot detector eye is then used in conjunction with a modified, but simplified version of the actual Dynamic C detection and tracking algorithm to control the Simbots' movements. The Java program outputs Simbot positional and rotational data to interpolator files that a Java program uses to generate a 3-dimensional animated scene replicating the Robot Wars competition.

Since the primary objective of this thesis is to accurately simulate the robot's optical detection system, several performance tests were conducted to measure the effectiveness of each simulated component in the system. The tests revealed that given a set of input parameters, the simulation was indeed effective in replicating the robot's optical detection system functionality. The significant component transfer functions have been implemented and placeholders are included for all other components. The implementation is completely modular so that transfer functions be easily altered in future simulations. The simulation is able to identify and replicate flaws in the original Dynamic C detection algorithm, mistuned (frequency) beacons and processors, low beacon battery supplies, and low processor battery supplies. The vital components of the detection system that were tested were the battery drainage of the beacon and processors, processor frequency response and the effects of nose discrimination and shading. Due to the fact that actual current supply measurements were not conducted, battery drainage behavior may not be realistic. Because of this and the fact that future robots may have

different battery/load characteristics, a battery loading matrix has been included in the simulation which can be easily used to input actual parameters. Tests have shown the simulation to be effective in providing an accurate depiction of how an actual robot behaves and performs.

In summary, the simulation implements the required physics for each of the pertinent components of the optical detection system. The overall simulation uses an object-oriented approach to model the behavior of two SE 3015 robots engaged in a series of detection, tracking, and attacking sequences. A 3-dimensional animation scene allows the user to visualize the behavior and motion of the two Simbots on a synthetic playing-field. The integrity, logic, and overall functionality of the Dynamic C detection algorithm and optical detection system have been preserved. As a result of these accomplishments, the goals and objective of this thesis and simulation have been met.

B. RECOMMENDATIONS FOR FUTURE WORK

Future work directly applicable to the optical detection system should include analysis and measurements of the currents supplied and various loading conditions for the Simbot processors. Verification and validation (V & V) of the simulated detection system with an actual robot is needed. V & V should include comparison of actual and simulated robot combat.

Future work on the Robot Wars simulation project should include improved physics for the various robot weapon systems and the drive train. implementation of various robot weapon systems to include the required physics for each. Most of the recommendation's in [Jones, 1997] are still important areas for continued work.

Some of the computer science related issues that must be addressed are the incorporation of a user-friendly Graphical User Interface (GUI) that allows the user to input robot parameters and interact with the simulation more easily. Furthermore, while in its present state, the simulation is capable of being accessed by anyone having an Internet connection and the appropriate VRML browser, its use of Java and VRML lays the foundation for the simulation to be fully networked. Appendix F describes how the simulation is set up and Appendix G describes how it may be accessed online.

APPENDIX A: JAVA SOURCE CODE

The simulation uses the code contained within **Simbot.java** to simulate a series of repetitive detection and tracking sequences between two SE 3015 robots. Angular velocites for each of the robot wheel speeds, wheel diameters, and acceleration/deceleration values are read in from the text file **robot.con**. The remainder of this appendix lists the source code, **Simbot.java** and the text file **robot.con**.

// File: Simbot.java
// Author: LT Wm. Ben McNeal
// Robot Wars Simulation Primary Program
// Operating Environment: Windows NT, Windows '95/'97, IRIX 6.*
// Compiler: JDK Version 1.1.3
// Description: This program simulates the movement of two Simbots engaged in a series of detection and tracking sequences
// Inputs: See Table 6-1
// Outputs: None
// Last compile/run date: 18 December 1997
// Warnings: None

```
import java.io.*;
import java.util.*;
//import vrml.*;
//import vrml.field.*;
//import vrml.node.*;

public class Simbot
{
      private static int EOF = 0;
      private static boolean TRACE = false;
      private static boolean posOpen = false;
      public static int pc = 0; //print counter so that at every .5
                                //secs output is printed to file and
                                //screen
      float robotX = 15.5f, robotZ = 15.5f,robotY = 0.425f;
      public float robotTheta = (float)Math.PI;

      float beacX =9.0f, beacZ = 9.0f;   //in feet
      private float bot_radius = .53125f; //in feet
      private float asgn_beac_freq = 16490f;
      private float beac_freq = asgn_beac_freq;
      private float filter_q = 100.0f;
      private float x = beac_freq / (2.0f * filter_q);
      private float sigma = (float)(Math.sqrt((double)(x*x)/.6931));

      private float MAX_SAT_DIST = 5.0f;
      private float MAX_DET_DIST = 20.0f;
      private int MAX_AD_INT = 4080;
      private int MIN_DET_INT = 48;
      private int MAX_NOISE_INT = 32;
      private int MIN_NOISE_INT = 16;
      private int Right_AD, Left_AD;

      private float NOSE_LENGTH = 1.5f;
      private float EYE_LENGTH = .344f;
      private float TOT_EYE_SEP = 0.5f;

      private float INIT_BCN_VOLT = 8.5f;
      private float INIT_EC_VOLT = 9.0f;

      private int timeStep = 100;     //  number of steps PER SECOND
      private float wheel_base,
```

```
            left_wheel_diam,
            right_wheel_diam;

      public float [][] speed_array; // 16 speeds for LEFT and RIGHT
wheels

      private float vel;

      private float accelDelta,
            drivenDecelDelta,
            freeDecelDelta;      //  constants for acceleration and
deceleration

      private static int  leftspeed, rightspeed;
      private static float duration = 0.1f;
      private static float time = 0.0f;
      private static float SCENE_TIME = 10.0f;
      private float numsteps, key;
      private float rps1, rps2;  // rps - revolution per second
      private float rpsorder1, rpsorder2;
      private float rpschange1, rpschange2;
      private float transientSteps1, transientSteps2;
      private float delta1, delta2;

      private PrintWriter ps_pos, ps_time, ps_angle, ps_key;
      private int i,j,k;

   public Simbot()
   {
      Float fltobj = new Float(0.0);

      try
      {
         BufferedReader in = new BufferedReader(new
FileReader("Robot.con"));

         in.readLine();
         in.readLine();
         String line = in.readLine();
         StringTokenizer t = new StringTokenizer(line, " \n\t\r");
         Float a = fltobj.valueOf(t.nextToken());
         accelDelta = a.floatValue();

         if (TRACE)
         System.out.println("accelDelta = " + accelDelta);

         in.readLine();
         line = in.readLine();
         t = new StringTokenizer(line, " \n\t\r");
         a = fltobj.valueOf(t.nextToken());
         drivenDecelDelta = a.floatValue();
         if (TRACE)
         System.out.println("drivenDecelDelta =" + drivenDecelDelta);

         in.readLine();
         line = in.readLine();
         t = new StringTokenizer(line, " \n\t\r");
```

```
        a = fltobj.valueOf(t.nextToken());
        freeDecelDelta = a.floatValue();
        if (TRACE)
        System.out.println("freeDecelDelta =" + freeDecelDelta);

        in.readLine();
        line = in.readLine();
        t = new StringTokenizer(line, " \n\t\r");
        a = fltobj.valueOf(t.nextToken());
        left_wheel_diam = a.floatValue();
        if (TRACE)
        System.out.println("left_wheel_diam =" + left_wheel_diam);

        a = fltobj.valueOf(t.nextToken());
        right_wheel_diam = a.floatValue();
        if (TRACE)
        System.out.println("right_wheel_diam =" + right_wheel_diam);

        in.readLine();
        line = in.readLine();
        t = new StringTokenizer(line, " \n\t\r");
        a = fltobj.valueOf(t.nextToken());
        wheel_base = a.floatValue();
        if (TRACE)
        System.out.println("wheel_base =" + wheel_base);

        in.readLine();
        line = in.readLine();
        speed_array = new float[16][2];
        for (int speed_num = 1; speed_num <= 15; speed_num++){
           t = new StringTokenizer(line, " \n\t\r");
           a = fltobj.valueOf(t.nextToken());
           vel = a.floatValue();
           speed_array[speed_num][0] = vel;
           a = fltobj.valueOf(t.nextToken());
           vel = a.floatValue();
           speed_array[speed_num][1] = vel;
           in.readLine();
           line = in.readLine();
          }//end outer for

        in.close();
        }
        catch  (IOException e)
        {
           System.out.print("Error: " + e);
           System.exit(1);
        }



        try
        {
              ps_pos = new PrintWriter
              (new BufferedOutputStream(new
FileOutputStream("botPos.dat")));
```

```
            ps_pos.println("Key Values for Position Interpolator");
            ps_pos.println(" ");
            ps_pos.println(" ");
         }
         catch  (IOException e)
         {
            System.out.print("Error: " + e);
            System.exit(1);
         }



         try
         {
               ps_key = new PrintWriter
               (new BufferedOutputStream(new
FileOutputStream("botKey.dat")));
               ps_key.println("Key for Interpolators");
               ps_key.println(" ");
               ps_key.println(" ");
         }
         catch  (IOException e)
         {
            System.out.print("Error: " + e);
            System.exit(1);
         }



         try
         {
               ps_angle = new PrintWriter
               (new BufferedOutputStream(new
FileOutputStream("botAngle.dat")));
               ps_angle.println("Key Values for Orientation
Interpolator");
               ps_angle.println("In file ");
               ps_angle.println(" ");
         }
         catch  (IOException e)
         {
            System.out.print("Error: " + e);
            System.exit(1);
         }



         if (TRACE){
            for (int i=0; i<16; i++)
               for (int j=0; j<2; j++)
                  System.out.println("speed_array[" + j + "][" + i + "]
= " + speed_array[i][j]);
         }

   } //  end of RobotClass definition
```

```
   public void get_command()
   {
      try
      {
         BufferedReader in = new BufferedReader
         (new FileReader("robot_tst.scr"));

         PrintWriter out = new PrintWriter
         (new BufferedWriter(new FileWriter("botPos.dat")));

         while (EOF != 1){
            readData(in);
              if (EOF != 1){

               System.out.println("leftspd = " + leftspeed + "
rightspd = " +
                                   rightspeed + "  duration = " +
duration);
               compute_pos(leftspeed, rightspeed);
              }
         }

         writeKey(ps_key);
         in.close();
      }

      catch  (IOException e)
      {
         System.out.print("Error: " + e);
         System.exit(1);
      }
    }



   public void run_program(int ad_rd0, int ad_rd1)
   {
      //System.out.println(ad_rd0 + " " + ad_rd1);
      if (time <= SCENE_TIME)
      {
         if (ad_rd0 <= 48 && ad_rd1 <= 48)
         {
            compute_pos(-8,8); //SLOW SPIN SEARCH
         }

         else if (ad_rd0 > 48 || ad_rd1 > 48)
         {
            if ((ad_rd0 >= 400 && ad_rd1 >=400) && (Math.abs(ad_rd0-
ad_rd1)<=50))
               compute_pos(8,8); //FINAL DETECTION MODE

            if (((ad_rd0<400 && ad_rd0>=250) && (ad_rd1<400 &&
ad_rd1>=250)) &&
            (Math.abs(ad_rd0-ad_rd1)<=50))
               compute_pos(10,10);//BOTH EYES EQUAL NOT QUITE THERE YET
```

```
            if (((ad_rd0<250 && ad_rd0>=100) && (ad_rd1<250 &&
ad_rd1>=100)) &&
            (Math.abs(ad_rd0-ad_rd1)<=50))
               compute_pos(12,12); //BOTH EYES EQUAL, APPROACHING

            if ((ad_rd0<100 && ad_rd1<100) && (Math.abs(ad_rd0-
ad_rd1)<=50))
               compute_pos(15,15); //BOTH EYES EQUAL, INITIAL
ACQUISITION

            if (ad_rd0-ad_rd1 > 2500)
               compute_pos(0,12); //SHARP LEFT TURN DIFF > 2500

            if (ad_rd0-ad_rd1>1000 && ad_rd0-ad_rd1<=2500)
                compute_pos(0,14);//SLOWER, SHARP LEFT TURN, DIFF >
1000

            if (ad_rd0-ad_rd1>500 && ad_rd0-ad_rd1<=1000)
               compute_pos(-10,10); //SLOW LEFT TURN, DIFF > 500

            if (ad_rd0-ad_rd1>50 && ad_rd0-ad_rd1<=500)
               compute_pos(-8,8);  //LEFT SEARCH TURN, DIFF > 50

            if (ad_rd1-ad_rd0>2500)
               compute_pos(12,0);  //SHARP RIGHT TURN DIFF > 2500

            if (ad_rd1-ad_rd0>1000 && ad_rd1-ad_rd0<=2500)
               compute_pos(14,0);//SLOWER, SHARP RIGHT TURN DIFF >1000

            if (ad_rd1-ad_rd0>500 && ad_rd1-ad_rd0<=1000)
               compute_pos(10,-10); //SLOW RIGHT TURN DIFF > 500

            if (ad_rd1-ad_rd0>50 && ad_rd1-ad_rd0<=500)
               compute_pos(8,-8); //RIGHT SEARCH TURN

         }

         else compute_pos(-8,8);
      }

      else writeKey(ps_key);
  }


   //Member Function:  compute_pos
   //Return Value:  void
   //Parameter:  left_wheel_speed and right_wheel_speed
   //Purpose:  This member function computes the new position of the
robot
   //
   private void compute_pos(int leftspeed, int rightspeed)
   {
      System.out.println(" ");
      System.out.println("LWS = " + leftspeed + " " + "RWS = " +
rightspeed);
           if (pc==0) {
```

```
            System.out.println(robotX + "  " + robotY + "  " + robotZ +
"  " +
                                robotTheta + " " + time + "  " + key);

          writePos(ps_pos,robotX,robotY,robotZ);
          writeAngle(ps_angle,robotTheta);
          }//end if

          pc++;

        /*   System.out.println("Robot Angle = " +
                                angle_norm((rad_to_deg(robotTheta)-
180.0f)));*/

           if ( leftspeed < 0 )
           {
              rpsorder1 = - speed_array[(-leftspeed)][0];
           }
           else
           {
              rpsorder1 = speed_array[leftspeed][0];
           }

                if ( rightspeed < 0 )
                {
                        rpsorder2 = - speed_array[(-rightspeed)][1];
                }
                else
                {
                        rpsorder2 = speed_array[rightspeed][1];
                }

         rpschange1 = rpsorder1 - rps1;
         rpschange2 = rpsorder2 - rps2;

        if ( ((rpsorder1 * rps1) <= 0 ) &&  rpsorder1 != 0 )
        {
            if (rpsorder1 > 0)
               delta1 = accelDelta / timeStep;
            else
               delta1 = -accelDelta / timeStep;
           transientSteps1 = rpschange1 / delta1;
        }
        else if ( rpschange1 > 0 )
        {
            delta1 = accelDelta / timeStep;
            transientSteps1 = rpschange1 / delta1;
        }
        else
        {
           if (rpsorder1 == 0)
              delta1 = -freeDecelDelta / timeStep;
           else
              delta1 = - drivenDecelDelta / timeStep;

           transientSteps1 = rpschange1 / delta1;
        }
```

```
        if ( ((rpsorder2 * rps2) <= 0 ) && rpsorder2 != 0 )
        {
          if ( rpsorder2 > 0 )
             delta2 = accelDelta / timeStep;
          else
             delta2 = - accelDelta / timeStep;

           transientSteps2 = rpschange2 / delta2;
        }
        else if ( rpschange2 > 0 )
        {
           delta2 = accelDelta / timeStep;
           transientSteps2 = rpschange2 / delta2;
        }
        else
        {
           if ( rpsorder2 == 0 )
              delta2 =  - freeDecelDelta / timeStep;
           else
              delta2 = - drivenDecelDelta / timeStep;

           transientSteps2 = rpschange2 / delta2;
        }

        numsteps = duration * timeStep;

        for (int step = 0; step < numsteps; step++)
        {
           rps1 += delta1;
           rps2 += delta2;

           float rad_of_curv = 0.0f;
           float bot_ang_vel = 0.0f;
           float deltaTheta = 0.0f;
           float deltaThetaHalf = 0.0f;
           float cons = 0.0f;
           float travel1 = 0.0f;
           float travel2 = 0.0f;

           float LEFT_WHEEL_SPD = rps1;
           float RIGHT_WHEEL_SPD = rps2;

          //System.out.println (bot_ang_vel);

           // Are the wheels traveling exactly the same (ie speed x dia
the same
           //   for each) ?  THIS IS NOT THE DISTANCE TRAVELED!!!!
           travel1 = LEFT_WHEEL_SPD * left_wheel_diam / 2;
           travel2 = RIGHT_WHEEL_SPD * right_wheel_diam / 2;

           if ( travel1 == travel2 )   //  traveling in a straight line
           {
              robotX += travel1 * (float)Math.sin( robotTheta ) /
(float) timeStep;
              robotZ += travel1 * (float)Math.cos( robotTheta ) /
(float) timeStep;
```

```
            deltaTheta = 0.0f;
         }
         else
         {
            rad_of_curv = (( LEFT_WHEEL_SPD * left_wheel_diam +
                          RIGHT_WHEEL_SPD * right_wheel_diam ) /
                          ( LEFT_WHEEL_SPD * left_wheel_diam -
                            RIGHT_WHEEL_SPD * right_wheel_diam )) *
                          ( wheel_base / 2.0f );
//if (TRACE)
//System.out.println("radius of curv = " + rad_of_curv );

            bot_ang_vel = ( LEFT_WHEEL_SPD * left_wheel_diam -
                    RIGHT_WHEEL_SPD * right_wheel_diam ) *
(float)Math.PI / wheel_base;
            if (TRACE)
            System.out.println("bot_ang_vel = " + bot_ang_vel);

            deltaTheta = bot_ang_vel / (float) timeStep;  // timeStep
is # of steps per second
            if (TRACE)
            System.out.println("deltaTheta = " + deltaTheta);

            deltaThetaHalf = deltaTheta / 2.0f;

            cons = 2.0f * rad_of_curv * (float)Math.sin(
deltaThetaHalf );
            if (TRACE)
            System.out.println("cons = " + cons);

            robotX += cons * (float)Math.sin( robotTheta +
deltaThetaHalf );
            robotZ += cons * (float)Math.cos( robotTheta +
deltaThetaHalf );

         }   //  end else ( travel1 != travel2 )

         robotTheta += deltaTheta;

         if ( step >= transientSteps1 )
         {
            delta1 = 0;
            rps1 = rpsorder1;
         }

         if ( step >= transientSteps2 )
         {
            delta2 = 0;
            rps2 = rpsorder2;
         }

      time += ( 1 / (float) timeStep ); ;

   }   //  end for ( int index = 0; index < numsteps; index++ )

      float angle = beac_rel_angle(robotX,robotZ,beacX,beacZ);
      System.out.println("angle = " + angle);
```

```
        float tend_angle =  180.0f + angle_norm((rad_to_deg(robotTheta)
- angle));

        float beac_dist =
beac_range(robotX,robotZ,beacX,beacZ,robotTheta);

        float bcn_int_fact = bcn_xmit_int(INIT_BCN_VOLT,time);
        beac_freq = bcn_flsh_freq(INIT_BCN_VOLT, time, beac_freq);

        float det_volt_fact = det_sply_volt(INIT_EC_VOLT,time);
        float freq_rsp_fact = filter(asgn_beac_freq,beac_freq,sigma);

        Left_AD =
bot_detector(beac_dist,beac_freq,tend_angle,bcn_int_fact,
                                    det_volt_fact,freq_rsp_fact, 0);

        Right_AD =
bot_detector(beac_dist,beac_freq,tend_angle,bcn_int_fact,
                                    det_volt_fact,freq_rsp_fact, 1);

        System.out.println("Left Eye = " + Left_AD + "     Right Eye =
" + Right_AD);

        System.out.println(robotX + "  " + robotZ + "  " + robotTheta +
" " + time);



        System.out.println("Range from beacon = " + beac_dist);
        System.out.println("Eye Angle = " +
angle_norm(rad_to_deg(robotTheta)));
        System.out.println("Relative angle of beacon to robot = " +
angle_norm(90.0f-angle));
        System.out.println("Angle between eyes and beacon = " +
tend_angle);

          System.out.println("PC = " + pc);

           if (  ( pc % 5 )==0 && pc !=0 ) {
            System.out.println(robotX + "  " + robotY + "  " + robotZ +
"  " +
                                robotTheta + " " + time + "  " + key);

          writePos(ps_pos,robotX,robotY,robotZ);
          writeAngle(ps_angle,robotTheta);
          }//end if

   run_program (Left_AD,Right_AD);
   }  //  end compute_pos



   //Member Function:  bcn_xmit_int
   //Return Value:  float ==> fraction of total intensity transmitted
by beacon
```

```
   //Parameter:  freq ==> frequency at which beacon oscillator is set
   //            volt ==> measured voltage of battery used to power
beacon oscillator
   //            time ==> amount of time gone by in scenario
   //Purpose:  This member function computes the fraction of total
intensity transmitted
   //          by the beacon
   //
   private float bcn_xmit_int(float volt, float time)
   {
      float volt_init = volt;
      float volt_drop = time * 5.56e-5f;
      float volt_final = volt_init - volt_drop;
      float int_fract = 1.0f;

      if (volt_final < 4.5f && volt_final >= 3.0f)
      {
        float volt_drop2 = 4.5f - volt_final;
        float int_delta = volt_drop2 * .1267f;
        int_fract  = int_fract - int_delta;
      }

      if (volt_final < 3.0f && volt_final >= 1.5f)
      {
         float volt_drop3 = 3.0f - volt_final;
         float int_delta = volt_drop3 * .533f;
         int_fract = .8f - int_delta;
      }

      if (volt_final < 1.5f)
      {
         int_fract = 0.0f;
      }

      if (TRACE)  System.out.println("Voltage Drop = " + volt_drop + "
Int. Fract = " +
                           int_fract);
      return int_fract;
   }



   //Member Function:  bcn_flsh_freq
   //Return Value:  float ==> voltage dependent flash frequency of
beacon
   //Parameter:  freq ==> frequency at which beacon oscillator is
initally set
   //            volt ==> measured voltage of battery used to power
beacon oscillator
   //            time ==> amount of time gone by in scenario
   //Purpose:  This member function computes the voltage dependent
flash frequency of
   //          the beacon
   //
   private float bcn_flsh_freq(float volt,float time, float freq)
   {
       float volt_init = volt;
```

```
      float volt_drop = time * 5.56e-5f;
      float volt_final = volt_init - volt_drop;

      float freq_drop = volt_drop / .0432f;
      float freq_final = freq - freq_drop;

      if(TRACE){
         System.out.println("Init.Volt. = " + volt_init + "  Final
Volt = " +
                            volt_final);
         System.out.println("Init. freq = " + freq + "  Final freq. =
" +
                            freq_final);
      }

      return freq_final;
   }


   //Member Function:  det_sply_volt
   //Return Value:  float ==> fraction of total detector intensity
received as
   //                         as result of its dependence on detector
supply voltage
   //Parameter:  volt ==> measured voltage of battery used to power
electrical circuit
   //            time ==> amount of time gone by in scenario
   //Purpose:  This member function computes the voltage dependent
intensity received by
   //          the detector
   //
   private float det_sply_volt(float volt, float time)
   {
      float volt_init = volt;
      float volt_drop = time * 5.56e-5f;
      float delta_int = volt_drop * .0391f;

      float int_fact = 1.0f - delta_int;

      return int_fact;
   }



   //Member Function:  filter
   //Return Value:  float ==> fraction of total detector intensity
received as
   //                         as result of its dependence on the
frequency
   //                         response of the electrical circuit
(bandpass filter)
   //Parameter:  asgn_beac_freq ==> assigned beacon frequency
   //            beac_freq      ==> reduced beacon freq as a result of
time and voltage
   //                               dependence
   //            sigma          ==> corrected characteristic width of
freq response curve
```

```
   //Purpose:  This member function computes the fraction of total
detector intensity
   //          received as as result of its dependence on the frequency
   //
   private float filter(float ass_freq, float curr_freq, float sigma)
   {
      float del_freq = Math.abs(ass_freq - curr_freq);
      float frac = (float)Math.exp((double)-(del_freq*del_freq)/(2.0f *
sigma * sigma));

      return frac;
   }



   //Member Function:  bot_detector
   //Return Value:  int ==> received intensity by detector in terms of
A/D integer
   //Parameter:  range,freq ==> Distance and angle  of beacon from
robot;
   //            and frequency of beacon
   //            MAX_AD_INT ==> max intensity detector capable of
producing due to
   //                           intensity received by beacon, detector
supply voltage,
   //                           and frequency filter function
   //Purpose:  This member function computes the received intensity by
detector in terms of A/D integer
   //
   public int bot_detector(float range,float freq,float
tend_angle,float bcn_int_fact,
                           float det_volt_fact, float
freq_rsp_fact,int lr_eye)
   {
      int rcvd_int = MIN_NOISE_INT ;
      int LEFT_EYE_INT = rcvd_int;
      int RIGHT_EYE_INT = rcvd_int;
      boolean reye_detect = false;
      boolean leye_detect = false;
      boolean value = false; //if computations are finished, value is
true and loop stops

if (TRACE) System.out.println("Bcn int fract = " + bcn_int_fact + "  EC
volt Fact = " +
                    det_volt_fact + "  Freq resp fact = " +
freq_rsp_fact);

      if (tend_angle <= 90.0f && tend_angle >=0.0f) leye_detect = true;
      if (tend_angle <= 360.0f && tend_angle >= 270.0f) reye_detect =
true;

      while ( (leye_detect || reye_detect) && !value )
      {
         if (range > MAX_DET_DIST)
         {
            leye_detect = reye_detect = false;
            break;
```

```
         }
         if (range <= MAX_SAT_DIST)
         {
            rcvd_int = (int)(bcn_int_fact * det_volt_fact *
freq_rsp_fact * MAX_AD_INT );
         }

         else rcvd_int = (int)(bcn_int_fact * det_volt_fact *
freq_rsp_fact * MAX_AD_INT *
                              ((MAX_SAT_DIST * MAX_SAT_DIST)/
(range*range)));

         rcvd_int = eye_ang_dep(rcvd_int, tend_angle);

       LEFT_EYE_INT = discriminate(rcvd_int,tend_angle,leye_detect);
       RIGHT_EYE_INT= discriminate(rcvd_int,tend_angle,reye_detect);

         if (LEFT_EYE_INT < MIN_NOISE_INT) LEFT_EYE_INT =
MAX_NOISE_INT;
         if (RIGHT_EYE_INT < MIN_NOISE_INT)RIGHT_EYE_INT=
MAX_NOISE_INT;

         value = true;
     }

     if (lr_eye == 0) rcvd_int = LEFT_EYE_INT;
     if (lr_eye == 1) rcvd_int = RIGHT_EYE_INT;

     return rcvd_int;
   }


   //Member Function:  eye_ang_dep
   //Return Value:  int ==> received intensity by detector in terms of
A/D integer
   //Parameter:  rcvd_int ==> on-axis intensity received by eyes,
dependent on range only
   //Purpose:  This member function applies an angular dependency to
the on-axis intensity
   //
   private int eye_ang_dep(int intensity, float angle)
      {
         angle = deg_to_rad(angle);
         intensity = (int)((float)intensity * (float)Math.cos(angle));
         return intensity;
      }
   //end eye_ang_dep



   //Member Function:  discriminate
   //Return Value:  int ==> received intensity by detector in terms of
A/D integer
   //Parameter:  rcvd_int ==> integer value of angle dependent
intensity received by eyes
   //            angle    ==> float value angle which beacon light
tends to eyes
```

```
   //            side     ==> boolean value that tells which side
beacon is detected on
   //Purpose:  This member function applies an eye nose and shadowing
dependency to the detector
   //          received intensity in order to give it left and right
eye discrimination
   private int discriminate(int rcvd_int, float angle, boolean side)
   {
      int intensity = rcvd_int;
      float theta_max = (float)Math.atan2((TOT_EYE_SEP/2.0f +
EYE_LENGTH),NOSE_LENGTH);
      float theta_min =
(float)Math.atan2((TOT_EYE_SEP/2.0f),NOSE_LENGTH);
      angle = deg_to_rad(angle);
      float DISCR_LENGTH = Math.abs(NOSE_LENGTH *
(float)Math.tan(angle));

      if (rad_to_deg(angle) >= 270.0f)
      {
          float T_MAX = theta_max;
        theta_max = (2.0f * (float)Math.PI) - theta_min;
        theta_min = (2.0f * (float)Math.PI) - T_MAX;
      }//end if

      if (!side)
      {
        if (rad_to_deg(angle) >= 270.0f)
          {
             if (angle <= theta_min) intensity = MAX_NOISE_INT;
             if (angle >= theta_max) intensity = rcvd_int;
             if (angle > theta_min && angle < theta_max)
                intensity = (int)(rcvd_int *
                ((1.0f + (TOT_EYE_SEP / 2.0f)/EYE_LENGTH) -
DISCR_LENGTH/EYE_LENGTH));
          }

          else
          {
             if (angle <= theta_min) intensity = rcvd_int;
             if (angle >= theta_max) intensity = MAX_NOISE_INT;
             if (angle > theta_min && angle < theta_max)
                intensity = (int)(rcvd_int *
                ((1.0f + (TOT_EYE_SEP / 2.0f)/EYE_LENGTH) -
DISCR_LENGTH/EYE_LENGTH));
          }
       }//end if (!side)

      //System.out.println("theta_max = " + rad_to_deg(theta_max) +
"theta_min =" +
      //                 rad_to_deg(theta_min));

      return intensity;
   }



   //Member Function:  beac_range
```

```
   //Return Value:  floating point value of range from beacon to robot
or robot eyes
   //Parameter:  botx,botz,beacx,beacz,bot_angle ==> Robot's and
Beacon's Position
   //Purpose:  This member function computes range of the beacon to the
robot from robot center
   //        and additionally computes the range of the beacon to the
robot's eyes
   //
   public float beac_range(float botx,float botz,float beacx,float
beacz,float bot_angle)
     {
        float range_beac;

        float eye_x = botx + bot_radius * (float)Math.sin(bot_angle);
        float eye_z = botz + bot_radius * (float)Math.cos(bot_angle);
        float range_eyes = ((eye_x-beacx)*(eye_x-beacx)) + ((eye_z-
beacz) * (eye_z-beacz));
        range_eyes = (float)Math.sqrt(range_eyes);
        //System.out.println("Eye range from beacon = " + range_eyes);

        range_beac = ((botx-beacx)*(botx-beacx)) + ((botz-beacz) *
(botz-beacz));
        range_beac = (float)Math.sqrt(range_beac);
        return range_eyes;
      }
   //end beac_range



   //Member Function:  beac_rel_angle;
   //Return Value:  float
   //Parameter:  botx,botz,beacx,beacz ==> Robot's and Beacon's
Position
   //Purpose:  This member function computes the relative angle of the
beacon
   //          to the robot
   //
   public float beac_rel_angle(float botx, float botz, float beacx,
float beacz)
      {
         float beac_theta;
         float x = botx - beacx;
         float z = botz - beacz;

         beac_theta = (float)Math.atan2(x,z);
         beac_theta = (rad_to_deg(beac_theta));
         if (beac_theta < 0) {
            beac_theta += 360.0;
         }
         return angle_norm(beac_theta);
      }
   //end beac


   //Member Function:  angle_norm;
   //Return Value:  float
```

```
   //Parameter:  theta ==> an angle in degrees
   //Purpose:  This member function normalizes and anlge so that its
value always
   //          falls between +0 and +360
   private float angle_norm(float theta)
      {
         while (theta < 0.0f || theta > 360.0f)
         {
            if (theta < 0.0f) theta   += 360.0f;
                if (theta > 360.0f) theta -= 360.0f;
         }

        return theta;
      }
   //end angle_norm;


   //Member Function:  rad_to_deg;
   //Return Value:  float
   //Parameter:  radians ==> an angle in radians
   //Purpose:  This member function converts an angle given in radians
to degrees
   //
   public float rad_to_deg(float radians)
      {
         float degrees;
         degrees = (float)((radians / (float)Math.PI) * 180.0);
         return degrees;
      }
   //end rad_to_deg



   //Member Function:  deg_to_rad
   //Return Value:  float
   //Parameter:  degrees ==> an angle in degrees
   //Purpose:  This member function converts an angle given in degrees
to radians
   //
   public float deg_to_rad(float degrees)
      {
         float radians;
         radians = (degrees * ((float)Math.PI / 180.0f));
         return radians;
      }
   //end deg_to_rad


  private void writePos(PrintWriter out, float x, float y, float z)
  {
     z = 18.0f - z;  //this compensates for coordinate system of VRML
playing field
     out.flush();
     i++;
     out.print(x + " " + " " + y + " " + z + ",");
     out.flush();
```

```
    if (i%3 == 0)
    out.println(" ");
    out.flush();
  }


  private void writeAngle(PrintWriter out, float theta)
  {
     theta = (2.0f * (float)Math.PI) - theta; //this compensates for
coord. sys. of
                                              // VRML playing field
     out.flush();

     k++;
     out.print("0.0 1.0 0.0  " + theta + ",  ");
     out.flush();
     if (k%3 == 0)
     out.println(" ");
     out.flush();
  }


   private void writeKey(PrintWriter out)
   {
      out.flush();

      float key_interval = 1.0f / ((float)SCENE_TIME / 0.5f);
      float key = 0.0f;
      key -= key_interval;

      for ( int i=0; i <= (int)(SCENE_TIME / 0.5f); i++)
      {
         key += key_interval;
         if (i == 0) key = 0;

         out.print(key + " ");
         if (i !=0 && i%5 == 0)
         {
            out.println(" ");
         }
      }
     out.flush();
   }


   public static void readData(BufferedReader in) throws IOException
   {
      Integer intobj = new Integer(0);

      String line = in.readLine();
      if (line == null)
      {
         EOF = 1;
         return;
```

```
        }

        StringTokenizer t = new StringTokenizer(line, " \n\t\r");

        leftspeed = intobj.parseInt(t.nextToken());
        System.out.println("leftspeed =" + leftspeed);
        rightspeed = intobj.parseInt(t.nextToken());
        System.out.println("rightspeed =" + rightspeed);
        duration = intobj.parseInt(t.nextToken());
        System.out.println("duration =" + duration);
    }



    public static void main(String[] args)
    {
        //PrintWriter VRML_file;

        Simbot BW1 = new Simbot();
        //BW1.get_command();
        BW1.run_program(16,16);
    }

}
```

// File: robot.con (ASCII)
// Author: LT Wm. Ben McNeal
// Subject: Simbot Configuration Program
// Operating Environment: NA
// Browser: NA
// Description: This program contains all of the physical characteristics for the Simbots (i.e., wheels speed, acceleration, deceleration)
// Inputs: None
// Outputs: None
// Last compile/run date: 18 December 1997
// Warnings: None

```
// Robot config - do not change comments around!  Bad things happen.
 // Acceleration constant rev per second per second
3.4
// Deceleration with one wheel driven rev per sec per sec -
drivenDecelDelta
0.35
// Deceleration with no wheels driven rev per sec per sec
0.84
// Wheel diameters in feet PORT (wheel 1) STBD (wheel 2)
.2742 .2755
// Wheel base feet
.750
// Angular velocities in revolutions per second, PORT STBD, speed 1 1
1.2 1.2
// Speed 2 2
0.2984 0.3182
// Speed 3 3
0.4096 0.4247
// Speed 4 4
0.5388 0.5557
// Speed 5 5
0.6526 0.6609
// Speed 6 6
0.7555 0.7711
// Speed 7 7
0.8656 0.8677
// Speed 8 8 0.9802
0.9573 0.9802
// Speed 9 9
1.0716 1.0779
// Speed a a
1.1714 1.2021
// Speed b b
1.2854 1.2893
// Speed c c
1.4331 1.4100
// Speed d d
1.5154 1.4302
// Speed e e
1.6279 1.6560
// Speed f f
```

1.7234 1.6555

APPENDIX B: INTERPOLATOR DATA OUTPUT FILES

The JAVA class, **Simbot.class** outputs the positional and rotational data for each Simbot into sets of output data files, which the graphics program uses to replicate the Simbots' behavior and movements. The output files consist of three text files for each Simbot. The first gives the keys for each of the Simbot positions listed in the previous two output files. The second gives the x, y, and z positions (key values) of the Simbot. The third gives the rotational position (key value) in terms of axis of rotation and the amount of rotation around that axis for each key. In VRML, the entire detection sequence scenario is scaled in time from 0.0 to 1.0. For example, if the entire scenario lasts 360 seconds (3 minutes), and we wish to use intervals of 0.5 seconds to animate the Simbot's movement, there would be 721 (360 * 0.5 + 1) equally spaced keys between 0 and 1 corresponding to the 721 0.5 second time intervals. This appendix lists examples of each of the output files for a 10 second scenario. They are listed in the JAVA source code as **bot1Pos.dat**, **bot2Pos.dat**, **bot1Angle.dat**, **bot2Angle.dat**, **bot1Key.dat**, and **bot2Key.dat**.

```
Key for Interpolators

0.0 0.05 0.1 0.15 0.2 0.25 0.3 0.35000002 0.40000004
0.45000005 0.5 0.5500001 0.6000001 0.6500001 0.7000001 0.7500001
0.80000013 0.85000014 0.90000015 0.95000017 1.000000


Key Values for Orientation (Rotation) Interpolator

0.0 1.0 0.0  3.1415927,0.0 1.0 0.0  3.3534286,
0.0 1.0 0.0  4.23458,  0.0 1.0 0.0  3.9552665,
0.0 1.0 0.0  3.772439, 0.0 1.0 0.0  3.8656433,
```

```
0.0 1.0 0.0  3.8196476,0.0 1.0 0.0  3.8038297,
0.0 1.0 0.0  3.8271668,0.0 1.0 0.0  3.850504,
0.0 1.0 0.0  3.873841, 0.0 1.0 0.0  3.8580232,
0.0 1.0 0.0  3.8813603,0.0 1.0 0.0  3.9046974,
0.0   1.0 0.0  3.9280345,  0.0 1.0 0.0  3.9513717,
0.0 1.0 0.0  3.9747088,
```

In this output file the key "0.0 1.0 0.0 3.3534286" is interpreted as no rotation about the x-axis or z-axis. The only rotation is about the y-axis, indicated by the 1.0 key value. The last value indicates the amount of rotation about the y-axis and is given in radians.

```
Key Values for Position Interpolator
15.5  0.425 2.5, 15.496904  0.425 2.527234,15.313581  0.425 2.7498045,
15.076797  0.425 2.8778868,14.956533  0.425 3.0596962,14.7258625  0.425
3.2966442,
14.454755  0.425 3.6119413,14.187247  0.425 3.93326,13.924856  0.425
4.258769,
13.667647  0.425 4.5883875,13.419444  0.425 4.9063673,13.159762  0.425
5.234042,
12.904133  0.425 5.546115,12.646233  0.425 5.856286,12.383597  0.425
6.1624784,
12.106548  0.425 6.475608,11.834393  0.425 6.7733717,11.557702  0.425
7.0669107,
11.274251  0.425 7.353915,10.986435  0.425 7.6365385,10.6898985 0.425
7.9100037
```

In this output file the key "15.5 0.425 2.5" is interpreted as a positions of 15.5 relative to the x-axis, 0.425 relative to the y-axis and 2.5 relative to the z-axis.

APPENDIX C: VRML SOURCE CODE

The following source code listed in this appendix, **Simbot.wrl**, is used to generate the 3-dimensional animated scene of the Simbot detection and tracking sequences. The contents of the interpolator output files listed in the previous appendix are "copied and pasted" into this code.

// File: Simbot.wrl (VRML file)
// Author: LT Wm. Ben McNeal
// Robot Wars Simulation Graphics Program
// Operating Environment: Windows NT, Windows '95/'97, IRIX 6.*
// Browser: CosmoPlayer 1.0
// Description: This program animlates the movement of two Simbots engaged in a series of detection and tracking sequences
// Inputs: See Appendix B
// Outputs: None
// Last compile/run date: 18 December 1997
// Warnings: None

```
#VRML V2.0 utf8


DEF RobotWorld Group{

   children [

   Background {

      skyColor [0.8 0.8 0.8,

                0.8 0.8 0.8,

                0.8 0.8 0.8

      ]

      skyAngle [1.309, 1.571]

   }

   NavigationInfo {
         type "EXAMINE"
   }
     Viewpoint{

         description "Front Center View"

         position 9.0 8.0 28.5

         fieldOfView  .916

      }

      Viewpoint{

         description "Back View"

         position 9.0 8.0 -11.0
```

```
   orientation 0.0 1.0 0.0 3.14
   fieldOfView .916
}
Viewpoint{
   description "Right Side View"
   position 28.0 8.0 10.0
   orientation 0.0 1.0 0.0 1.57
   fieldOfView .916
}
Viewpoint{
   description "Right Side Robot Level"
   position 18.0 4.0 10.0
   orientation 0.0 1.0 0.0 1.57
   fieldOfView .916
}
Viewpoint{
   description "Left Side View"
   position -10.0 8.0 10.0
   orientation 0.0 1.0 0.0 -1.57
   fieldOfView .916
}
Viewpoint{
   description "Left Side Robot Level"
   position -0.0 4.0 10.0
   orientation 0.0 1.0 0.0 -1.57
   fieldOfView .916
}
Viewpoint{
```

```
    description "Back Center Robot Level"
    position 9.0 4.0 0.0
    orientation 0.0 1.0 0.0 3.14
    fieldOfView .916
}
Viewpoint{
  description "Front Center Robot Level"
  position 9.0 4.0 14.0
  fieldOfView .916
}
Viewpoint{
    description "Front Right Corner Level"
    position 1.0 4.0 0.0
    orientation 0.0 1.0 0.0 3.84
    fieldOfView 1.047
}
Viewpoint{
    description "Front Left Corner Level"
    position 17.0 4.0 0.0
    orientation 0.0 1.0 0.0 2.443
    fieldOfView 1.047
}
Viewpoint{
    description "Beacon Display"
    position -30.0 9.0 9.25
    fieldOfView  .225
}
```

```
  Viewpoint{
     description "Robot Display"
     position 57.5 9.0 4.25
     fieldOfView .225
  }

Viewpoint{
     description "Gun Display"
     position 14.0 9.0 4.25
     fieldOfView .225
  }


   Transform{
      rotation 0.0 1.0 0.0 6.28
      children[
         Inline{ url "botfloor.wrl"}

      ]
   }

#ROBOT2
   Transform {
      translation 0.0 2.0 -1.0
      children [

         Viewpoint{

            description "Robot 2 Eye Level"

            position 0.0 1.0 -1.0

            fieldOfView .916

         }

         DEF BOT2MOV Transform{

           scale 0.0833 0.0833 0.0833

            translation 9.0 0.425 9.0

            rotation 0.0 1.0 0.0 3.14
```

```
            children[

               DEF Robot1 Inline{ url "robot2.wrl"}
            ]
         }
      ]
   }

#ROBOT1
   Transform {
      translation 0.0 2.0 -1.0
      children [

         Viewpoint{
            description "Robot 1 Eye Level"
            position 0.0 1.0 -1.0
            fieldOfView .916
         }
         DEF BOTMOV Transform{
           scale 0.0833 0.0833 0.0833
           translation 0.0 0.425 18.0
           #rotation 0.0 1.0 0.0 3.14
            children[
               DEF Robot1 Inline{ url "robot.wrl"}
            ]
         }
#Touch Sensor
   DEF TouchBot TouchSensor{}

# Animation Clock
   DEF Clock TimeSensor {
      cycleInterval 20.0
      stopTime 0.0
      startTime 1.0
      loop TRUE
   },

# Animation path
   DEF BotPath PositionInterpolator{
      key [ 0.0 0.05 0.1 0.15 0.2 0.25
            0.3 0.35000002 0.40000004 0.45000005 0.50000006
            0.5500001 0.6000001 0.6500001 0.7000001 0.7500001
            0.80000013 0.85000014 0.90000015 0.95000017 1.0000
      ]
      keyValue [15.5  0.425 2.5,15.496904  0.425 2.527234,15.313581
0.425 2.7498045,
15.076797  0.425 2.8778868,14.956533  0.425 3.0596962,14.7258625  0.425
3.2966442,
14.454755  0.425 3.6119413,14.187247  0.425 3.93326,13.924856  0.425
4.258769,
13.667647  0.425 4.5883875,13.419444  0.425 4.9063673,13.159762  0.425
5.234042,
12.904133  0.425 5.546115,12.646233  0.425 5.856286,12.383597  0.425
6.1624784,
12.106548  0.425 6.475608,11.834393  0.425 6.7733717,11.557702  0.425
7.0669107,
```

```
11.274251  0.425 7.353915,10.986435  0.425 7.6365385,10.6898985  0.425
7.9100037,
         ]
     }

  #Rotation Path
     DEF BotAngle OrientationInterpolator {
        key [0.0 0.05 0.1 0.15 0.2 0.25
             0.3 0.35000002 0.40000004 0.45000005 0.50000006
             0.5500001 0.6000001 0.6500001 0.7000001 0.7500001
             0.80000013 0.85000014 0.90000015 0.95000017 1.0000]

        keyValue [0.0 1.0 0.0  3.1415927,  0.0 1.0 0.0  2.9297569,  0.0
1.0 0.0  2.0486054,
                  0.0 1.0 0.0  2.327919,  0.0 1.0 0.0  2.5107465,  0.0
1.0 0.0  2.4175422,
                  0.0 1.0 0.0  2.4394214,  0.0 1.0 0.0  2.4552393,  0.0
1.0 0.0  2.4710572,
                  0.0 1.0 0.0  2.486875,  0.0 1.0 0.0  2.463538,  0.0
1.0 0.0  2.4793558,
                  0.0 1.0 0.0  2.4560187,  0.0 1.0 0.0  2.4326816,  0.0
1.0 0.0  2.4093444,
                  0.0 1.0 0.0  2.4251623,  0.0 1.0 0.0  2.4018252,  0.0
1.0 0.0  2.378488,
                  0.0 1.0 0.0  2.355151,  0.0 1.0 0.0  2.3318138,  0.0
1.0 0.0  2.3084767,
        ]
     }
       ]
    }
   ]
}

ROUTE TouchBot.isActive          TO Clock.set_enabled
ROUTE Clock.fraction_changed TO BotPath.set_fraction
ROUTE Clock.fraction_changed TO BotAngle.set_fraction
ROUTE BotPath.value_changed TO BOTMOV.set_translation
ROUTE BotAngle.value_changed TO BOTMOV.set_rotation
```

APPENDIX D: SAMPLE DYNAMIC C DETECTION AND TRACKING ALGORITHM

This appendix lists the detection and tracking algorithm portion of an actual Dynamic C program used in the SE 3015 Robot Wars competition. This code is discussed in Chapter IX.

// File: Excerpt from actual Dynamic C program
// Author: LT Wm. Ben McNeal
// Robot Wars Detection Algorithm
// Operating Environment: Windows
// Compiler: Dynamic C 1.1
// Description: This program is the detection and tracking logic for an SE 3015 robot
// Inputs: None
// Outputs: None
// Last compile/run date: September 1997
// Warnings: None

```
/* SLOW SPIN SEARCH */
  if (x<=48 && y<=48){
    strcpy(buf,"4fgr4rgr");
    platform();

                    }

  if (x>48 && y>48){

/* FINAL DETECTION MODE */
    if ((x>=400 && y>=400)  && (abs(x-y)<=50)){
       strcpy(buf,"8fsr8fsr");
       platform();

                                              }

/* BOTH EYES EQUAL NOTE QUITE THERE */
    if (((x<400 && x>=250) && (y<400 && y>=250))  && (abs(x-y)<=50)){
    strcpy(buf,"afgrafgr");
    platform();
    }

/* BOTH EYES EQUAL APPROACHING */
    if (((x<250 && x>=100) && (y<250 && y>=100))  && (abs(x-y)<=50)){
    strcpy(buf,"cfgrcfgr");
    platform();

}

/* BOTH EYES EQUAL INITIAL ACQUISITION */
    if ((x<100 && y<100)  && (abs(x-y)<=50)){
    strcpy(buf,"ffgrffgr");
    platform();
                                              }

/* SHARP LEFT TURN DIFF>2500 */
    if (x-y>2500){
    strcpy(buf,"9fsrcfgr");
    for(count=0;count<1000;count++);
    platform();
                }
```

```
/* SLOWER SHARP LEFT TURN DIFF>1000 */
    if (x-y>1000 && x-y<=2500){
    strcpy(buf,"9fsrefgr");
    for(count=0;count<1000;count++);
    platform();
                                }

/* SLOW LEFT TURN DIFF>500 */
    if (x-y>500 && x-y<=1000){
    strcpy(buf,"argrafgr");
    platform();
    for(count=0;count<1000;count++);
                               }


/* LEFT SEARCH TURN DIFF>250 */
    if (x-y>50 && x-y<=500){
    strcpy(buf,"8rgr8fgr");
    platform();
    for(count=0;count<1000;count++);
                              }

/* SHARP RIGHT TURN DIFF>2500 */
    if (y-x>2500){
    strcpy(buf,"cfgr9fsr");
    platform();
    for(count=0;count<1000;count++);
                }

/* SLOWER SHARP RIGHT TURN DIFF>1000 */
    if (y-x>1000 && y-x<=2500){
    strcpy(buf,"efgr9fsr");
    platform();
    for(count=0;count<1000;count++);
                               }

/* SLOW RIGHT TURN DIFF>500 */
    if (y-x>500 && y-x<=1000){
    strcpy(buf,"afgrargr");
    platform();
    for(count=0;count<1000;count++);
                               }

/* RIGHT SEARCH TURN DIFF>250 */
    if (y-x>50 && y-x<=500){
    strcpy(buf,"8fgr8rgr");
    platform();
    for(count=0;count<1000;count++);
                               }
                  }
         }
```

APPENDIX E: SIMBOT DETECTION AND TRACKING ALGORITHM

This appendix lists the Simbot detection and tracking algorithm portion of JAVA source code, **Simbot.class**. In the JAVA source code the function, ***run_program*** contains this algorithm. This code is discussed in Chapter IX.

// File: Excerpt from Simbot.java
// Author: LT Wm. Ben McNeal
// Simbot Detection and Tracking Algorithm converted from excerpt of Dynamic C Algorithm listed in Appendix D
// Operating Environment: Windows NT, Windows '95/'97, IRIX 6.*
// Compiler: JDK 1.1.3
// Description: This program simulates an actual algorithm used for robot detection and tracking
// Inputs: NA
// Outputs: None
// Last compile/run date: 18 December 1997
// Warnings: None

```
public void run_program(int ad_rd0, int ad_rd1)
   {
      //System.out.println(ad_rd0 + " " + ad_rd1);
      if (time <= SCENE_TIME)
      {
         if (ad_rd0 <= 48 && ad_rd1 <= 48)
         {
            compute_pos(-8,8); //SLOW SPIN SEARCH
         }

         else if (ad_rd0 > 48 || ad_rd1 > 48)
         {
            if ((ad_rd0 >= 400 && ad_rd1 >=400) &&
            (Math.abs(ad_rd0-ad_rd1)<=50))
               compute_pos(8,8); //FINAL DETECTION MODE

            if (((ad_rd0<400 && ad_rd0>=250) && (ad_rd1<400 &&
ad_rd1>=250)) &&
            (Math.abs(ad_rd0-ad_rd1)<=50))
               compute_pos(10,10); //BOTH EYES EQUAL NOT QUITE THERE
YET

            if (((ad_rd0<250 && ad_rd0>=100) && (ad_rd1<250 &&
ad_rd1>=100)) &&
            (Math.abs(ad_rd0-ad_rd1)<=50))
               compute_pos(12,12); //BOTH EYES EQUAL, APPROACHING

            if ((ad_rd0<100 && ad_rd1<100) && (Math.abs(ad_rd0-
ad_rd1)<=50))
               compute_pos(15,15); //BOTH EYES EQUAL, INITIAL
ACQUISITION

            if (ad_rd0-ad_rd1 > 2500)
               compute_pos(0,12); //SHARP LEFT TURN DIFF > 2500

            if (ad_rd0-ad_rd1>1000 && ad_rd0-ad_rd1<=2500)
                compute_pos(0,14);//SLOWER, SHARP LEFT TURN, DIFF >
1000

            if (ad_rd0-ad_rd1>500 && ad_rd0-ad_rd1<=1000)
```

```
            compute_pos(-10,10); //SLOW LEFT TURN, DIFF > 500

         if (ad_rd0-ad_rd1>50 && ad_rd0-ad_rd1<=500)
            compute_pos(-8,8);  //LEFT SEARCH TURN, DIFF > 50

         if (ad_rd1-ad_rd0>2500)
            compute_pos(12,0);  //SHARP RIGHT TURN DIFF > 2500

         if (ad_rd1-ad_rd0>1000 && ad_rd1-ad_rd0<=2500)
            compute_pos(14,0);//SLOWER, SHARP RIGHT TURN DIFF >1000

         if (ad_rd1-ad_rd0>500 && ad_rd1-ad_rd0<=1000)
            compute_pos(10,-10); //SLOW RIGHT TURN DIFF > 500

         if (ad_rd1-ad_rd0>50 && ad_rd1-ad_rd0<=500)
            compute_pos(8,-8); //RIGHT SEARCH TURN

      }

      else compute_pos(-8,8);
   }

}
```

APPENDIX F: SIMBOT SCRIPT

This appendix lists and discusses the Simbot script described in Chapter IX and used in the JAVA source code, **Simbot.class**.

As previously discussed, the simulation may be run in a mode where the target Simbot moves according to a user-defined script instead of responding to a detection and tracking algorithm. Instead of using the Java simulation function ***run_program***, the function ***run_script*** is used. The remainder of this appendix lists a sample of what a Simbot script would look like.

// File: Excerpt from Simbot.java
// Author: LT Wm. Ben McNeal
// Simbot Script
// Operating Environment: Windows NT, Windows '95/'97, IRIX 6.*
// Compiler: JDK 1.1.3
// Description: This program is a scripted Simbot target scenario
// Inputs: NA
// Outputs: None
// Last compile/run date: 18 December 1997
// Warnings: None

```
public void run_script(int ad_rd0, int ad_rd1)
   {
      if (time <= SCENE_TIME)
      {
         if (SCENE_TIME <= 5.0f)
         {
            compute_pos(8,8)
         }

         if (SCENE_TIME > 5.0f && SCENE_TIME <=10.0f)
         {
            compute_pos (-10,10)
         }

         if (SCENE_TIME > 10.0f && SCENE_TIME <=20.0f)
         {
            compute_pos(f,f)
         }
      }
   }
```

In this scenario, the Simbot moves for a total of 20.0 seconds. During each sampling instance the function is called and the target Simbot is issued a command from the script. During the first 5 seconds of the scenario, the target Simbot would be issued commands directing it to go straight, as indicated by ***compute_pos(8,8)***. In the next 5 seconds the target Simbot would spin on its axis as indicated by ***compute_pos(8,8)***. During the last 20 seconds the Simbot would move forward at full speed. Given the method's functionality and ease of use, it can be seen that there are endless possibilities for scripting one of the Simbots.

APPENDIX G: SIMULATION SETUP

This appendix describes the procedures for setting up and using the Simbot simulation. These steps assume that the user is familiar with compiling and running a program in Java and VRML. A VRML browser (plug-in) must be used in conjunction with Netscape *Navigator* or Internet *Explorer* to view the animated Simbot replication.

STEP 1: Decide which version of the simulation is to be run.

1. Mode 1 – Simbot 1 uses algorithm while Simbot 2 remains in fixed position on playing-field (Simbot1.java)
2. Mode 2 – Simbot 1 uses algorithm while Simbot 2 uses script (Simbot2.java)
3. Mode 3 – Both Simbots use algorithms (Simbot3.java)

STEP 2: Determine initial starting position and orientation of Simbots using playing-field coordinate system illustrated in Figure A-1.

STEP 3: Measure and record robot input parameters using Table 6-1.

STEP 4: Open appropriate simulation program (i.e., Simbot1.java).

STEP 5: Goto ***main*** function in program (at bottom).

STEP 6: Enter parameters in ***set*** function for appropriate robot (i.e., ***Simbot1.set(robotX,robotY,robotZ,...)***. If no values are entered, default values will be assigned to each parameter.

STEP 7: Save program.

STEP 8: Compile program.

STEP 9: Run program.

STEP 10: Open VRML file (Simbot.wrl).

STEP 11: Open key and data interpolator files and copy and paste data into appropriate section of VRML file. Appendix B shows the data in the interpolator files. In the VRML file

STEP 12: Save and close VRML file.

STEP 13: Open internet browser with appropriate VRML plug-in and load VRML file.

STEP 14: Watch Robot Wars!!!

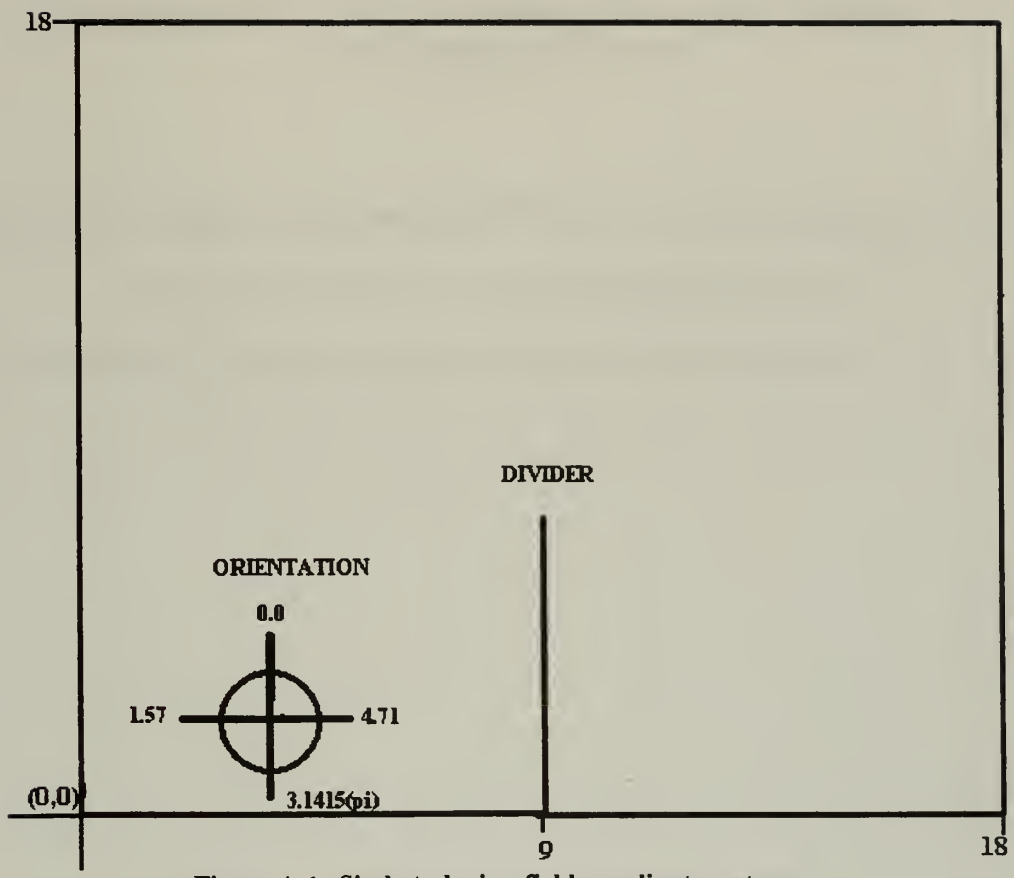

Figure A-1. Simbot playing-field coordinate system.

APPENDIX H: ONLINE RETRIEVAL OF THESIS AND SOURCE CODE

This thesis is online in both Postscript and HTML format at the following location:

http://www.stl.nps.navy.mil/~auv/cs4472/comsys.html

The source code used in the thesis is also available at the above address.

LIST OF REFERENCES

Brutzman, Don, CS 4202 (Introduction to Graphics) Course Notes, Naval Postgraduate School, Monterey, California, June 1997.

Ames, Andrea L., et al., VRML 2.0 Sourcebook, Second Edition, John Wiley and Sons, New York, 1997.

Carey, Rikk, et al., The Virtual Reality Modeling Language (VRML) Version 2.0 Specification, International Standards Organization/International Electrotechnical Commission (ISO/IEC) draft standard 14772, 1996.

Cornell, Gary and Horstmann, Cay S., Core Java, Second Edition, Sun Soft Press, Mountain View, California, 1997.

Halliday, David, et al., Fundamentals of Physics, Fourth Edition, John Wiley and Sons, New York, 1993.

Hofler, Tom, SE 3015 Course Notes, Naval Postgraduate School, Monterey, California, January 22, 1997.

Hofler, Richard M., SE 3015 Course Notes, Naval Postgraduate School, Monterey, California, January 22, 1997.

Lea, Rodger, et al., Java for 3D and VRML Worlds, New Rodgers Publishing, Indianapolis, 1996.

Naval Postgraduate School, Academic Year 1997, *Course Guide*, 1995, p. 307.

Simpson, Robert E., *Introductory Electronics for Scientists and Engineers*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

Zworld Engineering, *Dynamic C Application Frameworks*, revision 1.1, Davis, California, July 1995.

Zworld Engineering, *Tiny Giant C Programmable Miniature Controller Technical Reference Manual*, revision 1.1, Davis, California, November 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Rd., STE 0944
   Fort Belvoir, Virginia 22060-6218

2. Dudley Knox Library.....2
   Naval Postgraduate School
   Dyer Rd.
   Monterey, California 93943-5101

3. Professor William B. Maier.....1
   Chairman, Department of Physics
   Naval Postgraduate School (Code PH/We)
   Monterey, California 93943

4. Professor Gordon Schacher.....1
   Naval Postgraduate School (Code PH/Sq)
   Monterey, California 93943

5. Assistant Professor Don Brutzman.....1
   Chairman, Department of Physics
   Naval Postgraduate School (Code UW/Br)
   Monterey, California 93943

6. RADM Rodney Rempt.....1
   National Center 2 (PEO/TAD)
   2531 Jefferson Davis Highway
   Arlington, Virginia 22242-5170

7. Professor Tom Hofler.....1
   Naval Postgraduate School (Code PH/Ho)
   Monterey, California 93943

8. Associate Professor Don Walters.....1
   Naval Postgraduate School (Code PH/We)
   Monterey, California 93943

9. Associate Professor Scott Davis.....1
   Naval Postgraduate School (Code PH/Dv)
   Monterey, California 93943

10. Dr. Joe Cipriano.....1
National Center 2 (PEO/TAD-SE)
2531 Jefferson Davis Highway
Arlington, VA 22242-5170

11. Robert Wong.....1
Naval Postgraduate School
Monterey, California 93943

12. LT Wm. Ben McNeal.....2
5107 Tarryton Place
Jackson, Mississippi 39206